UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI

Gleyberson da Silva Andrade

Predição de Vulnerabilidades em Sistemas de Software Configuráveis: Uma abordagem baseada em Aprendizado de Máquina

São João del-Rei

2021

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI

Gleyberson da Silva Andrade

Predição de Vulnerabilidades em Sistemas de Software Configuráveis: Uma abordagem baseada em Aprendizado de Máquina

Dissertação apresentada como requisito para obtenção do título de mestre em Ciências no Curso de Mestrado do Programa de Pós Graduação em Ciência da Computação da UFSJ.

Orientador: Elder José Reioli Cirilo

Coorientador: Vinícius Humberto Serapilha Durelli

Universidade Federal de São João del-Rei – UFSJ

Mestrado em Ciência da Computação

São João del-Rei

2021

Ficha catalográfica elaborada pela Divisão de Biblioteca (DIBIB)
e Núcleo de Tecnologia da Informação (NTINF) da UFSJ,
com os dados fornecidos pelo(a) autor(a)

A553p

Andrade, Gleyberson da Silva.
Predição de Vulnerabilidades em Sistemas de Software Configuráveis : Uma abordagem baseada em Aprendizado de Máquina / Gleyberson da Silva Andrade ; orientador Elder José Reioli Cirilo; coorientador Vinícius Humberto Serapilha Durelli. -- São João del Rei, 2021.
74 p.

Dissertação (Mestrado - Ciência da Computação) -- Universidade Federal de São João del-Rei, 2021.

1. Machine Learning. 2. Vulnerabilidades. 3. Sistemas. I. Reioli Cirilo, Elder José, orient. II. Serapilha Durelli, Vinícius Humberto, co-orient. III. Título.

Gleyberson da Silva Andrade

Predição de Vulnerabilidades em Sistemas de Software Configuráveis: Uma abordagem baseada em Aprendizado de Máquina

Dissertação apresentada como requisito para obtenção do título de mestre em Ciências no Curso de Mestrado do Programa de Pós Graduação em Ciência da Computação da UFSJ.

Elder José Reioli Cirilo
Universidade Federal de São João del-Rei
Orientador

Vinícius Humberto Serapilha Durelli
Universidade Federal de São João del-Rei
Coorientador

Erick Galani Maziero
Universidade Federal de Lavras

Rafael Serapilha Durelli
Universidade Federal de Lavras

Diego Roberto Colombo Dias
Universidade Federal de São João del-Rei

São João del-Rei
2021

Agradecimentos

Primeiramente, agradeço a Deus por me dar forças e permitir a realização desse sonho. Agradeço aos meus pais, minha avó, minha irmã e minha namorada, que foram os pilares desta conquista e estiveram sempre ao meu lado, me apoiando e se orgulhando de mim a cada passo dado. Agradeço aos meus familiares e a todos os meus amigos, por sempre acreditarem que eu seria capaz dessa conquista. Agradeço também a meu orientador e aos meus professores, que com muita dedicação e paciência contribuíram para que este dia se tornasse realidade.

"Pra quem tem pensamento forte, o impossível é só questão de opinião."
(Chorão)

Resumo

Sistemas de *software* configuráveis oferecem uma grande variedade de benefícios, como suporte à configuração de comportamentos personalizados para necessidades específicas. Por outro lado, a complexidade induzida pelas opções de configuração no código-fonte requerem esforço adicional dos desenvolvedores ao adicionar ou editar instruções de código, levando a erros e à incidência de vulnerabilidades. Exercitar sistemas de *software* dinamicamente é uma tarefa cara, mas isso pode se tornar impraticável quando se trata de sistemas configuráveis, pois as variantes crescem exponencialmente conforme o número de variabilidades aumenta. Para contornar esse problema, podem ser empregadas heurísticas de amostragem para analisar apenas um subconjunto representativo das configurações do sistema, porém, a eficácia dessas heurísticas dependem da forma com que as amostras são selecionadas e às vezes, de fortes suposições de simplificação. Nossa proposta é recomendar um subconjunto de configurações que devem ser analisadas individualmente, utilizando estratégias baseadas em *Machine Learning*. Para tal, foram selecionadas 53 *features* referentes à Complexidade, Vulnerabilidade, Histórico e Desenvolvedor, para 11 projetos escritos em C. Esses projetos foram submetidos a execuções em diferentes cenários, tais como *Cross-validation* e *Cross-project-validation*, buscando reduzir o número de configurações recomendadas pela heurística LSA (Linear Sampling Algorithm), onde são combinados 3 algoritmos de amostragem: *One-enabled*, *One-disabled* e *Most-enabled-disabled*. Nossos resultados demonstram que é possível reduzir o tamanho da amostra de configurações, enquanto se mantém uma boa cobertura de detecção.

Palavras-chaves: *Machine Learning*, Vulnerabilidades, Sistemas configuráveis.

Abstract

Configurable software systems offer a variety of benefits such as supporting easy configuration of custom behaviours for distinctive needs. On the other hand, the complexity induced by configuration options in the source code requires additional effort from developers when adding or editing code statements, causing errors and incidence of vulnerabilities. Exercising software systems dynamically is an expensive task, but it can become impractical on configurable systems because variants grow exponentially as the number of variability increases. To cope with this situation, sampling heuristics can be employed to analyze only a representative subset of the system configurations, unfortunately, the effectiveness of vulnerability detection depends on how samples are selected and sometimes, strong simplifying assumptions. Our proposal is to recommend a subset of configurations that should be analyzed individually, using Machine Learning based strategies. Therefore, 53 resources related to Complexity, Vulnerability, History and Developer were collected, for 11 projects written in C. These projects were executed in some scenarios, such as Cross-validation and Cross-project-validation, seeking to reduce the configurations number recommended by LSA (Linear Sampling Algorithm) heuristic, where 3 sampling algorithms are combined: One-enabled, One-disabled and Most-enabled-disabled. Our results show that it is possible to reduce the configurations sample size, while maintaining a good detection coverage.

Key-words: Machine Learning, Vulnerabilities, Configurable systems.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

CWE	Common Weakness Enumeration
NVD	National Vulnerability Database
SMOTE	Synthetic Minority Over-sampling Technique
LSA	Linear Sampling Algorithm
AST	Abstract syntax tree
SQL	Structured Query Language
LDAP	Lightweight Directory Access Protocol
XSS	Cross-site scripting
USA	United States of America
FN	False negative
FP	False positive
TN	True negative
TP	True positive
TPR	True positive rate
FPR	False positive rate

Sumário

1 Introdução

Sistemas de *software* configuráveis oferecem uma grande variedade de benefícios, como suporte à configuração de comportamentos personalizados para necessidades específicas. Por outro lado, a complexidade induzida pelas opções de configuração no código-fonte tornam mais complexas as tarefas de manutenção e requer esforço adicional dos desenvolvedores ao adicionar ou editar instruções de código (Brabrand, 2013). A complexidade induzida pelas opções de configuração também faz com que os desenvolvedores cometam erros que levam à incidência de vulnerabilidades, já que o desenvolvimento desses códigos envolve raciocínios complexos de dependências de código configurável (Ferreira, 2016).

Alguns estudos já propuseram técnicas para detectar vulnerabilidades de segurança no código-fonte de sistemas de *software* (Sampaio; Garcia, 2016), mas normalmente utilizam apenas uma única variante por vez. No contexto de sistemas de *software* configuráveis, o número de variantes pode crescer exponencialmente com o número de opções de configuração, então analisar cada variante individualmente é inviável (Liebig, 2012). Como forma de contornar esse problema, podem ser empregadas heurísticas de amostragem para analisar apenas um subconjunto representativo das variantes do sistema (Medeiros, 2016).

Infelizmente, a eficácia da detecção de vulnerabilidade utilizando heurísticas de amostragem depende da forma com que as amostras são selecionadas, além de que muitas heurísticas de amostragem dependem de fortes suposições de simplificação. Essas suposições podem não ser realistas, nem práticas para a detecção de vulnerabilidades de segurança e podem não fornecer a cobertura de código desejada. A falta de heurísticas de amostragem adequadas pode levar a vulnerabilidades não detectadas e análises de código demoradas.

Neste trabalho, assumimos o desafio de recomendar um subconjunto de variantes que devem ser analisadas individualmente, utilizando uma estratégia baseada em *Machine Learning*. A proposta consiste em um estudo exploratório que visa investigar a aplicabilidade de técnicas de *Machine Learning* na detecção de vulnerabilidades em sistemas desenvolvidos em linguagem C, bem como avaliar se esses modelos de predição podem ser transferidos entre os projetos. Além disso, temos o objetivo de prover um maior entendimento sobre o uso de *Machine Learning* na criação de conjuntos de amostragem, buscando reduzir o número de variantes, enquanto se mantém uma boa cobertura de detecção.

Para a realização de tal avaliação, foram selecionados 11 projetos escritos em C: apache/httpd, cherokee/webserver, curl/curl, GNOME/libxml2, irssi/irssi, libav/libav, libexpat/libexpat, libssh2/libssh2, lighttpd/lighttpd1.4, openssl/openssl e torvalds/linux.

Uma característica fundamental desses projetos é que todos possuem diretivas de compilação condicional e repositório no *Git Hub* (GITHUB, 2021). Para cada um desses projetos, foi criada uma base de dados contendo todas suas funções. Para popular essa base, foram coletados dados referentes a 53 diferentes *features*, dividas em 4 diferentes categorias: complexidade, vulnerabilidade, histórico e desenvolvedor.

As *features* de complexidade foram coletadas utilizando a ferramenta Understand (SciTools, 2021). Essas *features* são responsáveis pelos dados cujas características são relacionadas à quantidade, como, por exemplo, número de linhas ou de determinada ocorrência. Ao todo, foram coletadas 16 *features* para compor essa categoria.

Já as *features* responsáveis pela categoria vulnerabilidade foram coletadas utilizando a ferramenta Joern (Joern, 2021). Para compor a base de dados, foram coletadas 22 *features* referentes a essa categoria. Essa categoria foi definida baseando-se no artigo de Xiaoning Du (Du, 2019) e representa características relacionadas a acessos à memória e utilização de ponteiros.

Para a realização da coleta das *features* de histórico foi realizada uma estratégia buscando no *log* de modificações do git, dados referentes a modificações no intervalo em que o projeto está versionado. Como exemplo, uma *feature* presente nessa categoria é o tempo de vida da função, representando a diferença de tempo entre o primeiro e último *commit* e que ela sofreu modificações. Das 53 *features* coletadas, 8 são responsáveis pela categoria histórico.

A última categoria é responsável pelas *features* de desenvolvedor. Essas *features* representam dados referentes aos autores que fizeram modificações nas funções do projeto no intervalo em que ele está versionado, de modo a mensurar a influência deles nessas funções. Um exemplo de *feature* desta categoria é o total de desenvolvedores que realizaram alguma modificação em determinada função. Ao todo, foram coletadas 7 *features* desta categoria.

Para apontar as funções vulneráveis presentes em cada um dos projetos, as bases de dados deles foram submetidas a 2 etapas. A primeira delas consistiu em buscar no *National Vulnerability Database* (NVD, 2021), notificações de vulnerabilidades referentes a *commits* desses projetos. Já a segunda etapa foi realizada de modo a complementar as incidências de vulnerabilidades, por meio de uma busca de padrões considerados vulneráveis, em mensagens de *commits* e pull requests do *Git Hub*.

Essas bases de dados foram submetidas a diferentes experimentos, de modo a encontrar o conjunto de *features* com maior capacidade de predição de vulnerabilidades, além de validar se um modelo treinado em determinado projeto apresenta um comportamento aceitável ao ser utilizado na predição das vulnerabilidades de outro projeto. Essas etapas foram realizadas utilizando técnicas de *cross-validation* e *cross-project-validation*,

além de um modelo treinado somente com o torvalds/linux, que por se tratar do *kernel* do sistema operacional do Linux, apresenta um número de funções muito maior que os demais.

Por fim, foram utilizados os resultados encontrados de modo a apontar as opções de configuração de *software* vulneráveis e assim, indicar conjuntos de variantes a serem testados. Esses conjuntos foram comparados, considerando a eficiência e esforço, aos conjuntos indicados pela heurística LSA (*Linear Sampling Algorithm*), onde são combinados 3 algoritmos de amostragem: *One-enabled*, *One-disabled* e *Most-enabled-disabled*.

1.1 Objetivos e Questões de Pesquisa

Com o intuito de nortear os experimentos, foram definidas as questões de pesquisa e o tipo de experimento a ser realizado. Considerando a vocação natural exploratória do estudo e com base na literatura (Easterbrook, 2008) (Woodside; Wilson, 2003), o Estudo de Caso Exploratório é o método de pesquisa mais adequado para servir de guia no decorrer do estudo e contribuir na formulação de novas hipóteses.

Foi utilizada a organização proposta pelo GQM (WOHLIN, 2012) para definir os objetivos do estudo de caso. Conforme o modelo de definição de objetivo proposto, o escopo deste estudo pode ser resumido da seguinte forma:

Analisar algoritmos de *Machine Learning*

para fins de avaliação

em relação à sua eficácia quanto à predição de vulnerabilidades

do ponto de vista do desenvolvedor

no contexto de sistemas configuráveis escritos em C.

Os objetivos acima mencionados podem ser enquadrados como as seguintes questões de pesquisa (RQs):

- RQ1: Quais *features* devem ser utilizadas na predição de vulnerabilidades de *software*?
- RQ2: Os modelos de predição podem ser transferidos entre projetos de *software*?
- RQ3: O modelo de predição tem um desempenho melhor do que as heurísticas de amostragem estado da arte?

1.2 Contribuições

Entre as principais contribuições desse trabalho de dissertação, destacam-se:

- Um estudo buscando identificar as *features* de *software* mais relevantes para a predição de vulnerabilidades;
- Um estudo da transferência de conhecimento (modelos) entre projetos;
- Um estudo sobre a criação de *samplings* para predição de vulnerabilidades em sistemas de *software* configuráveis.

1.3 Organização do Texto

O restante deste trabalho está organizado da seguinte forma: no Capítulo 2 é apresentada a fundamentação teórica e alguns conceitos importantes necessários que norteiam o presente trabalho; o Capítulo 3 apresenta a metodologia do trabalho, ou seja, todos os passos efetuados para a realização da estratégia proposta; o Capítulo 4 apresenta uma análise sobre os resultados obtidos; no Capítulo 5 é apresentada uma revisão da literatura, discutindo trabalhos existentes na área; por fim, no Capítulo 6, apresentamos uma breve conclusão de tudo que foi discutido em nosso trabalho.

2 Fundamentação Teórica

2.1 Sistemas de *software* Configuráveis

A reutilização de *software* é uma ambição de longa data da indústria de *software*. Desde as primeiras propostas relativas a componentes de *software* (McIlroy, 1968), era evidente a ambição dos engenheiros de *software* em compor sistemas da mesma forma que as crianças compõem peças de Lego (Ali Babar, 2010). Muitos sistemas de *software* modernos são projetados para serem altamente configuráveis, aumentando a flexibilidade, mas podendo tornar os programas difíceis de testar, analisar e entender (Reisner, 2010).

Sistemas de *software* Configuráveis possuem uma base de código genérica e um conjunto de mecanismos para implementar variações na estrutura e comportamento da base de código (Reisner, 2010). Esse tipo de sistema oferece uma variedade de benefícios, como suporte à configuração de comportamentos personalizados para necessidades específicas, tornando os sistemas de *software* extensíveis e portáteis. O paradigma (Figura 1) de desenvolvimento de sistemas configuráveis utiliza uma estratégia de reuso (Medeiros, 2014):

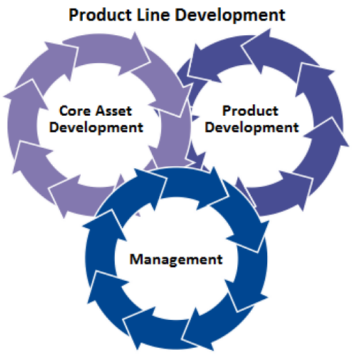

Figura 1 – Paradigma - Sistemas de *software* Configuráveis (Medeiros, 2014)

- Desenvolvimento de ativos: Nesta atividade são produzidos os ativos responsáveis pela base da linha de produto e sua capacidade de produção;
- Desenvolvimento de produto: Desenvolve o produto utilizando as saídas do desenvolvimento de ativos e os requisitos específicos de geração desse produto; e
- Gerenciamento: O desenvolvimento de ativos e as atividades de desenvolvimento de produtos são iterativas e devem ser gerenciadas.

Um conceito fundamental quando se trata de sistemas configuráveis é o de variabilidade. Variabilidade é a capacidade de alterar ou personalizar um sistema (Ali Babar, 2010). Quanto maior a variabilidade de um *software*, maior será a facilidade e o número de variantes que poderão ser gerados através dele (Pohl, 2005).

Para controlar as variações na estrutura, existem as opções de configuração, que podem ser definidas como unidades lógicas de comportamento, especificadas por conjuntos de requisitos funcionais ou não-funcionais (Ali Babar, 2010). Essas opções são responsáveis por definir as características do produto gerado, comumente chamado variante. Nesse contexto, os desenvolvedores usam diretivas condicionais, tais como #ifdef, #ifndef, #if e #elif, para marcar partes do código-fonte como opcionais. Essas diretivas tem como objetivo indicar quais conjuntos de instruções de código devem ou não estar presentes na variante gerada, considerando as opções de configuração selecionadas.

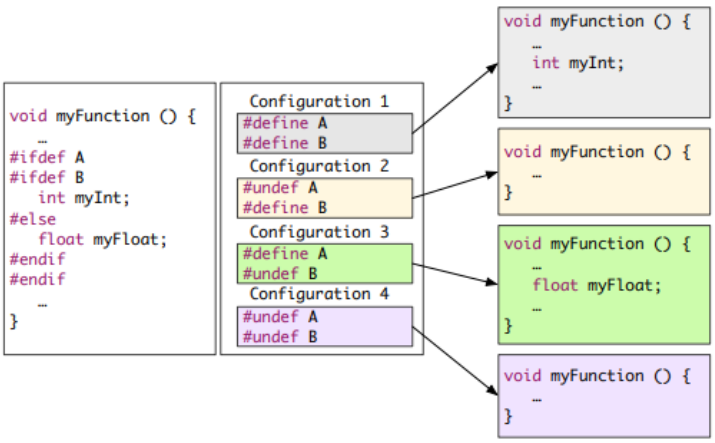

Figura 2 – Exemplo - Sistemas de *software* Configuráveis (Medeiros, 2014)

Como forma de contextualização, a Figura 2 apresenta um exemplo de código configurável. A primeira observação é quanto às 2 opções de configuração: A e B, responsáveis por controlar as diretivas de compilação condicional presentes na função *"myFunction"*. Nota-se também que existem 4 diferentes configurações, cada um responsável por gerar uma variante específica.

Analisando a primeira configuração, percebe-se que as opções de configuração A e B estão ativadas, já que possuem a diretiva #define. Isso garante que os escopos interiores às diretivas condicionais #ifdef A e #ifdef B serão incluídos na variante gerada. Isso pode ser confirmado ao se observar que a instrução *"int myInt;"* faz parte da variante gerada por essa configuração. De forma análoga, configurações com a diretiva #undef desativam a opção de configuração. É o caso, por exemplo, da terceira configuração, onde a opção de configuração A foi ativada, porém, a B não foi, fazendo com que somente a instrução *"float myFloat;"* fosse adicionada à variante gerada.

2.2 Vulnerabilidades

Diz-se vulnerável o que é suscetível de ser ferido ou atingido por doença (Michaelis, 2021). Em Computação, esse conceito de vulnerabilidade pode ser entendido como uma falha de segurança que pode fazer com que um invasor utilize um sistema de maneira diferente da pretendida pelo *designer* (Anley, 2007). Incidentes de segurança causados por ataques de computador, como hacking, negação de serviço, vírus e roubo de informações, podem ter um efeito negativo sobre a reputação de uma empresa, como perda de credibilidade, além de trazer enormes prejuízos financeiros para a organização (Candal-Vicente, 2017).

As vulnerabilidades de segurança podem ser divididas em vários tipos com base em diferentes critérios (Compuquip, 2021), tais como o local onde a vulnerabilidade existe, o que a causou ou como ela pode ser utilizada. Algumas categorias amplas desses tipos de vulnerabilidades incluem:

- Vulnerabilidades de rede: São problemas com o *hardware* ou *software* de uma rede que a expõem a uma possíveis acessos indesejados;
- Vulnerabilidades do sistema operacional: Essas são vulnerabilidades em um determinado sistema operacional que os *hackers* podem explorar para obter acesso ou causar danos a um ativo no qual o sistema operacional está instalado;
- Vulnerabilidades humanas: O elo mais fraco em muitas arquiteturas de segurança cibernética é o elemento humano. Os erros do usuário podem facilmente expor dados confidenciais, criar pontos de acesso exploráveis para invasores ou interromper sistemas; e
- Vulnerabilidades de processo: São vulnerabilidades criadas por processos específicos, como por exemplo o uso de senhas fracas.

Dentre os tipos de vulnerabilidades, existem categorias mais específicas, como as falhas de injeção, tais como injeção de *SQL*, de sistema operacional e de *LDAP*, que ocorrem quando dados não confiáveis são enviados para um interpretador como parte de um comando ou consulta. Os dados manipulados pelo atacante podem iludir o interpretador para que este execute comandos indesejados ou permita o acesso a dados não autorizados (OWASP, 2020).

Como forma de facilitar o uso eficaz de ferramentas que podem identificar, encontrar e resolver fragilidades em sistemas, existe o *Common Weakness Enumeration (CWE)* (CWE, 2021), um dicionário que lista as principais categorias de fragilidades e vulnerabilidades de *software*. O dicionário é mantido pela *MITRE Corporation* (Mitre, 2021) e pode ser acessado gratuitamente em todo o mundo. Anualmente, o *CWE* apresenta uma

lista demonstrativa dos 25 problemas mais comuns e impactantes ocorridos nos dois anos anteriores. A lista contém os tipos de fragilidades familiares aos especialistas em segurança. Entre os 10 principais da lista do ano de 2020 estão *cross-site scripting (XSS)*, injeção de *SQL*, validação de entrada imprópria, leitura fora dos limites e exposição de informações confidenciais a um ator não autorizado.

2.3 Detecção de Vulnerabilidades

Detectar vulnerabilidades é fundamental na inserção de medidas de segurança no desenvolvimento de sistemas. Existem muitas estratégias utilizadas para auxiliar nessa ação, desde a análise prévia até a análise baseada em testes. Embora este não seja o intuito deste estudo, a análise baseada em testes utiliza o conceito de gerar entradas para verificar a estabilidade do *software*, é o caso da técnica *Fuzzing* (Li, 2018), onde são geradas entradas massivas (normais e anormais) para todas as entradas do sistema a ser analisado, para então, tentar detectar exceções alimentando as entradas geradas para os aplicativos de destino, monitorando os estados de execução.

Além da análise baseada em testes, existem também as análises feitas no próprio sistema, como por exemplo a análise de fluxo e a análise baseada em padrões de código. A análise de fluxo tem a capacidade de seguir os caminhos de determinado dado através de variáveis do código desde a origem até seu término, com isso é possível detectar pontos vulneráveis no sistema, já que um dado informado pelo usuário pode ser uma instrução maliciosa.

A análise baseada em padrões de código parte do princípio que todo código é padronizado da mesma forma, já que toda invocação de método ou declaração de variável possui a mesma sintaxe. Partindo desse conceito, é possível definir quais características ou instruções dos métodos são responsáveis por inserir vulnerabilidades no sistema. Com isso, a análise é feita no código-fonte do sistema em busca de padrões que representam essas instruções, assumindo como vulnerável cada ponto encontrado.

2.4 Algoritmos de *Machine Learning*

Machine Learning é um campo de pesquisa na interseção de estatística, inteligência artificial e ciência da computação e consiste em extrair conhecimento dos dados (Müller; Guido, 2016). Existem várias classes de aprendizagem em *Machine Learning*, sendo as mais comuns descritos por Ayon Dey (Dey, 2016):

- Aprendizagem supervisionada: São aqueles algoritmos que precisam de assistência externa. O conjunto de dados de entrada é dividido em conjunto de dados de treinamento e teste. O conjunto de dados tem uma variável de saída que precisa ser

prevista ou classificada. Existem dois tipos principais de problemas de aprendizado de máquina supervisionado (Müller; Guido, 2016):

  - Classificação: Na classificação, o objetivo é prever um rótulo de classe, que é um conjunto finito e predefinido de classes. A classificação às vezes é separada em classificação binária, que é o caso especial de distinguir exatamente duas classes, e classificação multiclasse, que é a classificação entre mais de duas classes;
  - Regressão: Para tarefas de regressão, o objetivo é prever um número contínuo ou um número de ponto flutuante em termos de programação (ou número real em termos matemáticos). Prever a renda anual de uma pessoa baseando-se em características dessa pessoa é um exemplo de uma tarefa de regressão.

- Aprendizagem não supervisionada: São os algoritmos onde quando novos dados são introduzidos, usam os recursos aprendidos anteriormente para reconhecer a classe dos dados. Uma característica dessa categoria de aprendizagem é que não existem rótulos nos dados e as tarefas comuns são agrupamento e regras de associação.

- Aprendizagem semi-supervisionada: Nessa categoria, o conjunto de dados contém exemplos rotulados e não rotulados. Normalmente, a quantidade de exemplos não rotulados é muito maior do que o número de exemplos rotulados. O objetivo de um algoritmo de aprendizagem semi-supervisionado é o mesmo que o objetivo do algoritmo de aprendizagem supervisionada. A motivação em se utilizar essa categoria de aprendizagem é que muitos exemplos não rotulados pode ajudar o algoritmo de aprendizagem a encontrar um modelo melhor (BURKOV, 2019).

- Aprendizagem por Reforço: Aprendizado por reforço é um subcampo do aprendizado de máquina onde a máquina é capaz de perceber o estado de um ambiente como um vetor de recursos e executar ações. Diferentes ações trazem recompensas e penalidades, além de mover a máquina para outro estado do ambiente. O objetivo de um algoritmo de aprendizagem por reforço é aprender uma política que leva o vetor de características de um estado como entrada e produz uma ação ideal para executar nesse estado. A ação é ótima se maximizar a recompensa média esperada (BURKOV, 2019).

- Aprendizagem Multitarefa: Tem o objetivo de ajudar outros algoritmos a ter um desempenho melhor. Quando esses algoritmos são aplicados em uma tarefa, armazenam as etapas de como se resolve o problema ou como chega a uma conclusão específica, para que sejam utilizadas para encontrar a solução de outro problema ou tarefa semelhante.

- Aprendizagem em Conjunto: Quando vários algoritmos individuais são combinados para formar apenas um algoritmo. Existem dois tipos de aprendizado em conjunto:
  - *Boosting*: É uma técnica de aprendizagem em conjunto que é usada para diminuir o viés e a variância.
  - *Bagging*: É aplicada onde a precisão e estabilidade de um algoritmo de aprendizado de máquina precisa ser aumentada;
- Aprendizagem por redes neurais: É derivada do conceito biológico de neurônios, a camada de entrada recebe a entrada, assim como os dendritos. A camada oculta processa a entrada, como a soma e o axônio. Finalmente, a camada de saída envia a saída calculada, como os terminais dendríticos; e
- Aprendizagem baseada em instância: O algoritmo aprende um determinado tipo de padrão e tenta aplicar o mesmo padrão aos dados recém-alimentados. É um tipo de algoritmo *"preguiçoso"*, que espera que os dados do teste cheguem e então age sobre eles junto com os dados de treinamento. A complexidade do algoritmo de aprendizagem aumenta com o tamanho dos dados.

Cada uma dessas classes possuem algoritmos específicos responsáveis por realizar a predição de bases de dados. Neste estudo, foram utilizados dois algoritmos de árvores (*Random Forest, J48*), um algoritmo de aprendizagem em conjunto (*Gradient Boosting*), um algoritmo de aprendizagem por redes neurais (*Multilayer Perceptron*) e um algoritmo probabilístico (*Naive Bayes*). Esses algoritmos serão descritos a seguir:

- *Random Forest*: Método de aprendizado que gera várias árvores de decisão durante o tempo de treinamento. Cada árvore fornece um rótulo de classe e o classificador seleciona o rótulo da classe que possui o modo de saída das classes por árvores individuais (Alenezi; Abunadi, 2015);
- *Árvore de Decisão (J48)*: Árvore de Decisão é um classificador na forma de uma estrutura em árvore. É um modelo preditivo que decide o valor dependente de uma nova amostra com base em diversos valores de atributo dos dados existentes. Cada nó interno na árvore representa um atributo único enquanto nós de folha representam rótulos de classe. As árvores de decisão classificam cada exemplo, começando na raiz da árvore e movendo-a até alcançar um nó folha (Alenezi; Abunadi, 2015);
- *Gradient Boosting*: É um algoritmo classificado como aprendizagem de conjunto. Foi proposto pela primeira vez por Freund e Schapire (1996) para problemas de classificação e era conhecido como AdaBoost. Desde então, tem sido usado em muitos campos e mostra acurácia preditiva semelhante ou superior aos métodos tradicionais, tanto em problemas de classificação quanto de regressão (Gonzalez-Recio, 2013);

- *Multilayer Perceptron*: Solução de rede neural proposta por Rumelhart, Hinton e McClelland, é um dos métodos de aprendizado de máquina mais comumente usados (Marsland, 2014). O aprendizado nesse tipo de rede é na maioria das vezes feito através do algoritmo de retropropagação do erro. *Multilayer Perceptron* é uma rede neural com uma ou mais camadas ocultas e um número indeterminado de neurônios. Cada camada da rede tem uma função específica, sendo que a camada de saída recebe os estímulos da camada intermediária e constrói a resposta; e
- *Naive Bayes*: Os classificadores *Naive Bayes* são uma família de classificadores bastante semelhantes aos modelos lineares, no entanto, eles tendem a ser ainda mais rápidos no treinamento. A razão pela qual os modelos *Naive Bayes* são tão eficientes é que eles aprendem parâmetros olhando para cada recurso individualmente e coletam estatísticas simples por classe de cada recurso (Müller; Guido, 2016).

Um modelo de classificação binária tem como objetivo decidir em qual classe uma instância pertence dentre duas classes possíveis. Existem 4 cenários possíveis em uma predição, falso negativo (FN), falso positivo (FP), verdadeiro negativo (TN) e verdadeiro positivo (TP), sendo estes comumente representados através de uma matriz de confusão.

Como forma de facilitar a compreensão de cada um destes cenários, um modelo de classificação de uma função vulnerável pode ser usado como exemplo, sendo as classes positiva e negativa, a presença e a ausência da vulnerabilidade, respectivamente. Então, os cenários ilustrados na matriz de confusão podem ser descritos da seguinte maneira:

- TP: Classificar uma função como vulnerável e ela é vulnerável;
- TN: Classificar uma função como não vulnerável e ela não é vulnerável;
- FP: Classificar uma função como vulnerável, mas ela não é vulnerável; e
- FN: Classificar uma função como não vulnerável, mas ela é vulnerável;

Existem diversas métricas responsáveis por avaliar a eficiência de um modelo de classificador baseando-se nos resultados da matriz de confusão. A Acurácia (2.1) é definida como a soma do número de verdadeiros positivos e verdadeiros negativos dividido pelo número total de exemplos (Marsland, 2014). Essa métrica representa quantos exemplos foram de fato classificados corretamente, independente da classe.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.1}$$

A Precisão (2.2) é a razão de exemplos classificados corretamente como positivos, pelo número real de exemplos positivos, enquanto Recall é a proporção do número de

exemplos classificados como positivos, pelo número de exemplos que foram classificados como positivos (Marsland, 2014). A Precisão é usada como uma métrica de desempenho quando o objetivo é limitar o número de falsos positivos, já Recall é usada quando é importante evitar falsos negativos (Müller; Guido, 2016).

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

Embora precisão e recall sejam métricas muito importantes, a análise do valor de apenas uma delas pode não fornecer o resultado completo da análise do modelo (Müller; Guido, 2016). Uma maneira de resumi-los é através da métrica f1-score (2.4), ou f-measure, sendo obtida através do cálculo da média harmônica entre a precisão e o recall.

$$F - Score = \frac{precision * recall}{precision + recall} * 2 \tag{2.4}$$

Visto que se pode usar essas métricas para avaliar classificadores, também é possível compará-los. Uma forma de realizar essa comparação é através da curva ROC (Marsland, 2014). Essa curva é representada através de um gráfico da porcentagem de verdadeiros positivos no eixo Y em relação a falsos positivos no eixo X (Figura 3).

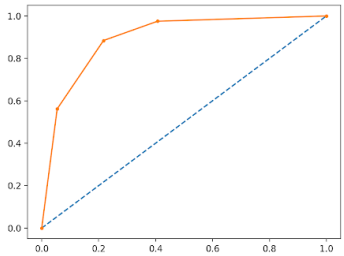

Figura 3 – Curva ROC

Quanto mais próxima a curva estiver do canto superior esquerdo, melhor é a predição do modelo (Müller; Guido, 2016). Com base nessa afirmação, a métrica AUC é responsável por calcular a área sob a curva ROC, seguindo a premissa de que quanto maior essa área, melhor será o modelo.

Embora AUC seja a métrica de precisão diagnóstica mais comumente usado, o Índice de Youden também é frequentemente usado na prática (Stefan, 2011). Esse índice é calculado através das taxas de verdadeiros positivos (*sensitivity* ou *TPR* (2.5)), sendo a probabilidade que um verdadeiro positivo seja classificado como positivo; e taxas de

verdadeiros negativos (*specificity* ou *TNR* (2.6)), sendo a probabilidade que um verdadeiro negativo seja classificado como negativo.

$$TPR = \frac{TP}{TP + FN} \tag{2.5}$$

$$TNR = \frac{TN}{TN + FP} \tag{2.6}$$

O Índice de Youden tem uma característica atrativa não presente na AUC: fornecer um critério para encontrar o valor limite para o qual as taxas de TP e TN são maximizadas 2.7.

$$Youden = max(TPR + TNR - 1) \tag{2.7}$$

Outras taxas relevantes, além da TPR e TNR, são: a taxa de falsos positivos (FPR (2.8)), que é a probabilidade que um exemplo seja classificado como positivo, quando o valor verdadeiro for negativo; e a taxa de falsos negativos (FNR (2.9)), que é a probabilidade de que um verdadeiro positivo não seja atingido pelo teste.

$$FPR = \frac{FP}{FP + TN} \tag{2.8}$$

$$FNR = \frac{FN}{FN + TP} \tag{2.9}$$

2.5 Algoritmos de *Sampling*

Exercitar sistemas de *software* dinamicamente é uma tarefa cara, mas isso pode se tornar impraticável quando se trata de sistemas configuráveis, pois as variantes crescem exponencialmente conforme o número de variabilidade aumenta (Medeiros, 2016). Portanto, em geral, os métodos de análise clássicos não escalam na prática e se tornou necessária uma nova categoria de análises com base na variabilidade. O objetivo é reduzir o esforço de análise explorando as semelhanças entre as variantes. Em contraste, as estratégias de *sampling* lidam com o crescimento exponencial de variantes, sugerindo que, em vez de analisar todas as variantes, deve-se selecionar um subconjunto representativo para ser analisado individualmente, um por um.

Existem vários algoritmos que visam gerar o melhor subconjunto: aquele que maximiza nossas chances de encontrar *bugs* e possui o menor número de variantes. Dois algoritmos de *sampling* clássicos são: *One-enabled* e *One-disabled*. O *One-enabled* produz o subconjunto selecionando todas as variantes que possuem apenas uma variabilidade

ativada. Por outro lado, *One-disabled* procede selecionando todas as variantes que têm apenas uma variabilidade desabilitada. As amostragens resultantes são do tamanho N em termos do número de variabilidade (Medeiros, 2016).

LSA (Medeiros, 2016) é outro algoritmo de *sampling* que gera amostras de variantes cujos tamanhos crescem linearmente em termos do número de variabilidade, em contraste com algoritmos *t-wise* (Oh, 2019). O algoritmo LSA combina três algoritmos de *sampling*: *One-enabled*, *One-disabled* e *Most-enabled-disabled*. Este último consiste em gerar duas variantes, a primeira com todas as variabilidades ativadas, e a segunda com todas as variabilidades desativadas. Com isso, o algoritmo de *sampling LSA* seleciona as variabilidades de forma linear, ou seja, seleciona 2n + 2 variabilidades, onde n é o número de opções de variabilidade. Conforme avaliado em (Medeiros, 2016), o LSA tem apresentado bons resultados em termos de capacidades de detecção de *bugs* e tamanhos de conjuntos de amostras.

3 Projeto dos Estudos Empíricos

Neste capítulo, são descritos os passos executados na condução de 3 estudos buscando responder às questões de pesquisa. O primeiro deles utiliza uma estratégia de *cross-validation* para investigar a aplicabilidade de técnicas de *Machine Learning* na detecção de vulnerabilidades. O segundo, através de uma estratégia de *cross-project-validation*, avalia se os modelos de predição podem ser transferidos entre projetos. Por fim, o último estudo avalia o uso de *Machine Learning* na criação de conjuntos de amostragem.

Ao todo, seis passos resumem toda a estratégia utilizada, iniciando pela seleção dos projetos, seguido da criação da base de dados original, coleta dos dados e definição das funções vulneráveis. Então, são detalhados os passos da execução dos estudos cujo objetivo é responder às questões de pesquisa *RQ1* e *RQ2*. Por fim, é descrita a estratégia de geração de variantes e os passos responsáveis por responder à questão de pesquisa *RQ3*.

3.1 Seleção dos projetos

O primeiro passo é a seleção dos projetos a serem analisados. Ao todo, foram selecionados 11 do GitHub (GITHUB, 2021), escritos predominantemente em C. Abaixo são apresentadas curtas descrições dos projetos selecionados:

- apache/httpd (APACHE/HTTPD, 2020): É um servidor do tipo HTTPD (*Hypertext Transfer Protocol Daemon*), compatível com o protocolo HTTP versão 1.1;
- cherokee/webserver (CHEROKEE/WEBSERVER, 2020): É um servidor WEB de código aberto inovador, rico em recursos, extremamente rápido e fácil de configurar, projetado para a próxima geração de aplicativos da web altamente seguros e simultâneos;
- curl/curl (CURL/CURL, 2020): Curl é uma ferramenta de linha de comando para transferir dados especificados com sintaxe de URL;
- GNOME/libxml2 (GNOME/LIBXML2, 2020): Libxml2 é o analisador XML C e o kit de ferramentas desenvolvido para o projeto GNOME;
- irssi/irssi (IRSSI/IRSSI, 2020): Irssi é um cliente de chat modular mais conhecido por sua interface de usuário em modo de texto;
- libav/libav (LIBAV/LIBAV, 2020): Libav é uma coleção de bibliotecas e ferramentas para processar conteúdo multimídia, como áudio, vídeo, legendas e metadados relacionados;

- libexpat/libexpat (LIBEXPAT/LIBEXPAT, 2020): É um analisador XML orientado a fluxo, onde os manipuladores são registrados com o analisador antes de iniciar a análise;
- libssh2/libssh2 (LIBSSH2/LIBSSH2, 2020): Libssh2 é uma biblioteca que implementa o protocolo SSH2;
- lighttpd/lighttpd1.4 (LIGHTTPD/LIGHTTPD1.4, 2020): Lighttpd é um servidor web seguro, rápido, compatível e muito flexível que foi otimizado para ambientes de alto desempenho;
- openssl/openssl (OPENSSL/OPENSSL, 2020): OpenSSL é um kit de ferramentas de código aberto robusto, de nível comercial e cheio de recursos para o protocolo *Transport Layer Security* (TLS), anteriormente conhecido como protocolo *Secure Sockets Layer* (SSL); e
- torvalds/linux (TORVALDS/LINUX, 2020): É um núcleo monolítico de código aberto para sistemas operacionais tipo UNIX.

Como forma de padronizar a extração dos dados, foram coletados *commits* até o fim do ano de 2020, exceto para o libav/libav, cujo último *commit* aconteceu em abril de 2019 e o torvalds/linux, que por ser um projeto com lançamentos periódicos e um grande volume de contribuições, foram coletados até a *release* 5.10, lançada em dezembro de 2020. A Tabela 1 apresenta as datas das versões para as quais os projetos foram coletados, além de descrever as características de cada um deles.

Tabela 1 – Sistemas - Características

Sistema	**Número de funções**	**Número de commits**	**Data**
apache/httpd	27.519	32.452	Dec 19, 2020
cherokee/webserver	3.292	5.846	Sep 19, 2020
curl/curl	5.879	26.875	Dec 31, 2020
GNOME/libxml2	8.083	5.043	Dec 18, 2020
irssi/irssi	5.393	6.259	Dec 31, 2020
libav/libav	23.983	45.232	Apr 16, 2019
libexpat/libexpat	1.356	3.187	Dec 29, 2020
libssh2/libssh2	1.079	2.128	Dec 18, 2020
lighttpd/lighttpd1.4	3.258	3.789	Oct 20, 2020
openssl/openssl	26.978	28.351	Dec 31, 2020
torvalds/linux	1.056.822	995.984	Dec 13, 2020

3.2 Criação da base de dados inicial

O próximo passo foi a extração de todas as funções de cada um dos projetos. Para a execução dessa etapa foram definidos 3 atributos capazes de identificar individualmente uma função, são eles: caminho do arquivo onde a função se encontra, nome e lista dos tipos dos parâmetros dela.

Com isso, foi criado um *dataset* para cada projeto. Cada *dataset* possui 3 colunas, representando o caminho do arquivo, nome da função e se a mesma é vulnerável. Essas colunas receberam os nomes: *File, Function e Vulnerable*, respectivamente. Nessa etapa, todas as linhas do *dataset*, em *Machine Learning* comumente denominadas *Samples* (Müller; Guido, 2016), são inicializadas como não vulneráveis.

3.3 Coleta dos dados

Nesta etapa, foi realizada a coleta dos dados referentes às características de cada uma das funções. Essas características (*Features*) (Müller; Guido, 2016), são representadas por colunas no *dataset*. Como forma de simplificar a seleção, foram definidas subetapas, onde cada é responsável pela coleta de conjuntos específicos de *features*. Ao todo, foram definidas 4 subetapas de coleta: 1) Coleta das *features* de complexidade, utilizando a ferramenta *Understand* (SciTools, 2021); 2) Coleta das *features* de histórico; 3) Coleta das *features* de desenvolvedor; 4) Coleta das *features* de vulnerabilidade, utilizando a ferramenta *Joern* (Joern, 2021).

3.3.1 Coleta - *Features* de complexidade de código

Para a coleta das *features* de complexidade, foi utilizado o *Understand* [1], um ambiente de desenvolvimento integrado que permite a análise estática de código, por meio de uma variedade de ferramentas capazes de calcular métricas relacionadas ao código analisado. *Understand* suporta uma grande variedade de linguagens, tais como Python, Java, C# e até mesmo Assembly, além das linguagens que são o foco deste estudo: C/C++.

Essa ferramenta possibilita ao usuário selecionar o nível de profundidade da análise, indo desde métricas por arquivo, ou até mesmo por funções, além de possibilitar a seleção de quais métricas devem ser extraídas. No campo da segurança de *software*, especialistas argumentam que a complexidade é inimiga da segurança, sendo a causa de vulnerabilidades sutis que são difíceis de testar (Younis, 2016). Esse é o motivo principal que levou à seleção das 16 *features* de complexidade a seguir:

- CountLineCodeExe: Número de linhas contendo código-fonte executável;

[1] https://www.scitools.com/

- CountInput: Número de subprogramas de chamada mais variáveis globais lidas;
- CountLine: Número total de linhas;
- CountLineBlank: Número de linhas em branco;
- CountLineCode: Número de linhas contendo código-fonte;
- CountLineCodeDecl: Número de linhas contendo código-fonte declarativo;
- CountOutput: Número de chamadas de subprogramas, somado ao conjunto de variáveis globais;
- CountPath: Número de caminhos possíveis, sem contar saídas ou gotos anormais;
- CountStmt: Número de declarações;
- CountStmtDecl: Número de declarações declarativas;
- CountSemicolon: Número de ponto-e-vírgulas;
- Cyclomatic: Complexidade ciclomática;
- CyclomaticModified: Complexidade ciclomática modificada;
- CyclomaticStrict: Complexidade ciclomática estrita;
- Knots: Medida de saltos sobrepostos;
- MaxNesting: Nível máximo de aninhamento de construções de controle.

Finalizada a extração dos valores para cada uma das funções, foram criadas *features* no *dataset* inicial para representá-los.

3.3.2 Coleta - *Features* de histórico

Nesta subetapa foram coletadas as *features* que refletem características do histórico das funções. Como justificativa para a importância de *features* de histórico, podem ser citados três pontos destacados por Bosu (Bosu, 2014): 1) Enquanto a maioria dos tipos de vulnerabilidade são frequentemente corrigidos, algumas vulnerabilidades críticas, como, por exemplo, acesso indevido e *deadlock*, tem alta taxa de abandono, além disso, 2) a probabilidade de que um *patch-set* contenha uma vulnerabilidade aumenta proporcionalmente ao número de linhas alteradas e 3) arquivos modificados são mais propensos a conter vulnerabilidades que novos arquivos.

Para a realização desta coleta foi utilizado o *framework* PyDriller (Spadini, 2021). Esse framework auxilia desenvolvedores na análise de repositórios Git, possibilitando a

extração de informações sobre *commits*, desenvolvedores, arquivos modificados, *diffs* e código-fonte. A estratégia dessa subetapa consistiu em obter quatro valores utilizados na geração das *features* de histórico: 1) lista contendo o número de linhas adicionadas por *commit*; 2) lista contendo o número de linhas removidas por *commit*; 3) dicionário contendo a *hash* de cada *commit*, além de dados do desenvolvedor (nome, número de linhas modificadas e o momento em que foi realizado o *commit*), utilizados na geração das *features* de desenvolvedor e serão mencionados na próxima subetapa; e 4) lista de *timestamps* por *commit*.

Para facilitar a compreensão, abaixo (Listagem 3.2) é mostrado o resultado obtido para o *git log* exibido na Listagem 3.1. Esta Listagem exibe o *log* de modificações para uma função escolhida aleatoriamente no projeto torvalds/linux (*update_sched_clock*, presente no arquivo *kernel/time/sched_clock.c*). Uma observação é que cada *hash* presente no dicionário "*commits*"contém dados do desenvolvedor, mas foram ocultados, já que não são utilizados nessa subetapa:

Listing 3.1 – Git log - Resultado

```
COMMIT_INFO->13dbeb384d2d3aa555ea48d511e8cb110bd172e0|1427441638|Daniel Thompson
diff --git a/kernel/time/sched_clock.c b/kernel/time/sched_clock.c
--- a/kernel/time/sched_clock.c
+++ b/kernel/time/sched_clock.c
@@ -116,20 +114,20 @@
 static void notrace update_sched_clock(void)
 {
        unsigned long flags;
        u64 cyc;
        u64 ns;
        struct clock_read_data *rd = &cd.read_data;

-       cyc = rd->read_sched_clock();
+       cyc = cd.actual_read_sched_clock();
        ns = rd->epoch_ns +
             cyc_to_ns((cyc - rd->epoch_cyc) & rd->sched_clock_mask,
                       rd->mult, rd->shift);

        raw_local_irq_save(flags);
        raw_write_seqcount_begin(&cd.seq);
        rd->epoch_ns = ns;
        rd->epoch_cyc = cyc;
        raw_write_seqcount_end(&cd.seq);
        raw_local_irq_restore(flags);
 }


COMMIT_INFO->cf7c9c170787d6870af54684822f58acc00a966c|1427441637|Daniel Thompson
diff --git a/kernel/time/sched_clock.c b/kernel/time/sched_clock.c
--- a/kernel/time/sched_clock.c
+++ b/kernel/time/sched_clock.c
@@ -81,19 +116,20 @@
 static void notrace update_sched_clock(void)
 {
        unsigned long flags;
        u64 cyc;
        u64 ns;
+       struct clock_read_data *rd = &cd.read_data;

-       cyc = read_sched_clock();
-       ns = cd.epoch_ns +
-               cyc_to_ns((cyc - cd.epoch_cyc) & sched_clock_mask,
```

```
-                       cd.mult, cd.shift);
+       cyc = rd->read_sched_clock();
+       ns = rd->epoch_ns +
+            cyc_to_ns((cyc - rd->epoch_cyc) & rd->sched_clock_mask,
+                      rd->mult, rd->shift);

        raw_local_irq_save(flags);
        raw_write_seqcount_begin(&cd.seq);
-       cd.epoch_ns = ns;
-       cd.epoch_cyc = cyc;
+       rd->epoch_ns = ns;
+       rd->epoch_cyc = cyc;
        raw_write_seqcount_end(&cd.seq);
        raw_local_irq_restore(flags);
}
```

Listing 3.2 – Git log - Exemplo de extração - *Features* de histórico

```
{
    "added_lines": [1, 7],
    "removed_lines": [1, 6],
    "commits": {"13dbeb384d2d3aa555ea48d511e8cb110bd172e0": {},
                "cf7c9c170787d6870af54684822f58acc00a966c": {}},
    "time": [1427441638, 1427441637]
}
```

Baseando-se nos valores obtidos, foram definidas as *features* a serem calculadas para cada uma das funções, são elas:

- MaxAddedLines: Maior número de linhas adicionadas por *commit*. No exemplo da Listagem 3.2, corresponde ao valor 7;
- MaxRemovedLines: Maior número de linhas removidas por *commit*. No exemplo da Listagem 3.2, corresponde ao valor 6;
- MedianAddedLines: Mediana do número de linhas adicionadas. No exemplo da Listagem 3.2, corresponde ao valor 4;
- MedianRemovedLines: Mediana do número de linhas removidas. No exemplo da Listagem 3.2, corresponde ao valor 3,5;
- Lifetime: Tempo de vida da função, calculada pela diferença entre o maior e o menor valor da lista "*time*". Uma observação é que se a função ainda existir no momento da coleta, ou seja, não tiver sido removida em nenhum *commit*, essa métrica é calculada pela diferença entre o *timestamp* do momento da coleta e o menor valor da lista "*time*". No exemplo da Listagem 3.2, corresponde ao valor 1, desde que no momento da coleta a função não exista mais e assim, o valor é calculado somente com os presentes na lista "*time*";

- CodeChurn: Representa o quanto a função foi modificada em um determinado período de tempo, ou seja, a soma do número de linhas adicionadas e removidas ao longo dos *commits*. No exemplo da Listagem 3.2, corresponde ao valor 15;
- RelativeChurn: Representa a razão entre o CodeChurn e o número total de linhas da função, representado pela da métrica CountLineCode obtido na seção anterior (3.3.1). No exemplo da Listagem 3.2, onde o valor de CountLineCode é 15, essa métrica tem como valor o resultado de 15/15, ou seja, 1; e
- CommitsCount: Número total de *commits*, no exemplo da Listagem 3.2, corresponde ao valor 2.

Da mesma forma que na subetapa anterior, foram criadas *features* no *dataset* inicial para representar os dados extraídos.

3.3.3 Coleta - *Features* de desenvolvedor

Nesta etapa foram selecionadas as *features* dos desenvolvedores das funções. A extração dessas *features* depende diretamente da realização da subetapa anterior, onde cada *hash* presente no dicionário "*commits*", representado na Listagem 3.2, possui os dados do desenvolvedor referente ao *commit* que ela representa.

Dois fatores apontados em (Bosu, 2014) justificam a coleta de *features* representando características dos desenvolvedores. O primeiro deles é que a maioria das vulnerabilidades são introduzidas por autores que fazem as maiores alterações no código. O segundo está relacionado à experiência do desenvolvedor, onde autores menos experientes têm 1,8 a 24 vezes mais probabilidade de introduzir vulnerabilidades.

Nessa subetapa foram coletados os dados de todos os desenvolvedores responsáveis pela criação de pelo menos um *commit* no projeto. Como mencionado acima, esses dados foram coletados utilizando o *framework* PyDriller. A Listagem 3.3 exibe os dados extraídos do exemplo representado pela Listagem 3.1.

Listing 3.3 – Git log - Exemplo de extração - *Features* de desenvolvedor

```
{
    "commits": {"13dbeb384d2d3aa555ea48d511e8cb110bd172e0": {"time": 1427441638,
                                                             "dev": "Daniel Thompson",
                                                             "added_lines": 1,
                                                             "removed_lines": 1},
                "cf7c9c170787d6870af54684822f58acc00a966c": {"time": 1427441637,
                                                             "dev": "Daniel Thompson",
                                                             "added_lines": 7,
                                                             "removed_lines": 6}}
}
```

A estratégia da coleta de dados dos desenvolvedores se resumiu em percorrer os dados de todas as funções e para cada *commit*, armazenar o registro do desenvolvedor responsável por ele. Caso já tenha sido salvo um registro desse desenvolvedor, ou seja, esse desenvolvedor já tenha sido o responsável de algum *commit* coletado anteriormente, os dados foram combinados. Por exemplo, o desenvolvedor Daniel Thompson, responsável pelos 2 *commits* representados na Listagem 3.1, teria seus dados armazenados de forma similar à representação ilustrada na Listagem 3.4.

Listing 3.4 – Dados armazenados do desenvolvedor

```
{
    "Daniel Thompson": {
        "added_lines": 8,
        "removed_lines": 7,
        "functions": 1,
        "commits": [
            "13dbeb384d2d3aa555ea48d511e8cb110bd172e0",
            "cf7c9c170787d6870af54684822f58acc00a966c"
        ],
        "commit_times": [
            1427441638,
            1427441637
        ]
    }
}
```

Após o processo de geração do dicionário, foram calculados mais dois dados para cada desenvolvedor. O primeiro representa o quanto o desenvolvedor é influente no projeto: *ratio_total_contribution_dev*. Esse dado foi calculado pela razão entre o valor presente no campo "*functions*"e o número total de funções do projeto. O segundo representa o intervalo de tempo que o usuário contribuiu para o projeto: *uptime_dev*. Esse dado foi calculado pela diferença entre o maior e menor valor presente no campo "*commit_times*".

O próximo passo foi a definição de *features* para as funções baseando-se nos dados dos desenvolvedores. No total, sete *features* foram responsáveis pela representação desses dados:

- RatioTotalContributionDev: Representa o ratio_total_contribution_dev médio dos desenvolvedores da função;
- AvgTotalCommitDev: Representa a média do número de *commits* realizados pelos desenvolvedores da função;
- AvgTotalModifiedLinesDev: Representa a média da soma de linhas adicionadas e removidas pelos desenvolvedores da função;
- AvgUptimeDev: Representa a média do tempo de contribuição dos desenvolvedores da função;

- TotalDevFunc: Representa o total de desenvolvedores que realizaram alguma modificação na função;
- MaxPondDevFunc: Representa o maior número de linhas modificadas em um só *commit* dividido pelo total de linhas modificas; e
- MinPondDevFunc: Representa o menor número de linhas modificadas em um só *commit* dividido pelo total de linhas modificas.

O passo final dessa subetapa foi a criação de colunas no *dataset* para armazenar as *features* de desenvolvedor coletadas.

3.3.4 Coleta - *Features* de vulnerabilidade

A maioria dos tipos críticos de vulnerabilidades em programas C/C++ são direta ou indiretamente causados por erros de gerenciamento de memória e/ou falta de verificação em algumas variáveis sensíveis, como, por exemplo, ponteiros (Du, 2019).

As *features* coletadas nessa subetapa foram baseadas no trabalho de Xiaoning Du (Du, 2019), que utiliza o *Joern* para coletar *features* e com base nelas, identificar funções potencialmente vulneráveis. A última subetapa é responsável pela coleta das *features* de vulnerabilidade, por meio da ferramenta *Joern* [2]. *Joern* analisa e representa toda a base de código em um grafo de propriedades, permitindo que características do código sejam extraídas usando consultas de pesquisa formuladas em uma linguagem baseada em Scala.

Para representar a classe "vulnerabilidade", foram definidas e coletadas as seguintes *features* para as funções de cada um dos projetos:

- ParametersCount: Representa o número de parâmetros;
- DeclarationsCount: Representa o número de variáveis declaradas;
- LoopsCount: Representa o número de estruturas de repetição;
- NestedLoops: Representa o número de estruturas de repetição aninhadas;
- UsesInLoops: Representa o número de variáveis utilizadas em estruturas de repetição;
- LoopDeclarations: Representa o número de variáveis declaradas em estruturas de repetição;
- AccessPointers: Representa o número de operações de acesso a membros de estruturas de ponteiro (a->b);

[2] https://joern.io/

- AddressPointers: Representa o número de operações de acesso a endereços de estruturas de ponteiro (&a);
- ParametersPointers: Representa o número de parâmetros que são ponteiros;
- AllUsesPointers: Representa o número de usos de ponteiros no escopo da função;
- AssigmentsPointers: Representa o número de atribuições a ponteiros;
- EqualityPointers: Representa o número de ponteiros envolvidos em operações de igualdade;
- OperationPointers: Representa o número de ponteiros envolvidos em operações de adição e subtração;
- IfWithoutElse: Representa o número de estruturas condicionais simples;
- UsesInIfStatements: Representa o número de variáveis utilizadas em estruturas condicionais;
- DeclarationsInIfStatements: Representa o número de variáveis declaradas em estruturas condicionais;
- ControlStatements: Representa o número de estruturas de controle;
- NestedControlStatements: Representa o número de estruturas de controle aninhadas;
- UsesInControlStatements: Representa o número de variáveis utilizadas em estruturas de controle;
- ControlStatementsDeclarations: Representa o número de variáveis declaradas em estruturas de controle;
- Controls: Representa o número de estruturas de controle dependentes de controle; e
- ControlStructsControlingInputData: Representa o número de estruturas de controle dependentes de dados.

Ao fim dessa subetapa, assim como em todas as anteriores, foram criadas colunas no *dataset* para armazenar as *features* extraídas.

3.4 Definição das funções vulneráveis

Esta seção é responsável por detalhar os passos utilizados na identificação das funções ditas vulneráveis na base de dados. Duas subetapas são responsáveis pela definição das funções vulneráveis, são elas a busca por vulnerabilidades notificadas no repositório *NVD* (NVD, 2021) e a busca por padrões nas mensagens de *commits* e *pull requests*.

3.4.1 Análise do *National Vulnerability Database (NVD)*

NVD é o repositório do governo dos EUA de dados de gerenciamento de vulnerabilidades. Esse repositório possui bancos de dados que armazenam falhas de *software* relacionadas à segurança no decorrer dos anos. O primeiro passo da estratégia da busca pelas funções vulneráveis foi realizar o *download* de todos os dados das vulnerabilidades notificadas entre os 2002 a 2020.

Com isso, para cada um dos projetos analisados, foi criado um arquivo para armazenar os *commits* considerados vulneráveis. Para preenchê-los, os arquivos do *NVD* foram percorridos em busca de ocorrências e caso encontrasse, o *commit* em questão era salvo. Então, para cada um desses *commits*, foi executado o comando descrito na Listagem 3.5, listando todas as modificações presentes nele.

Listing 3.5 – Git diff - Comando

```
git diff commit~1 commit
```

O próximo passo foi a realização de um *parse* da saída desse comando, atualizando a entrada desse *commit* no arquivo com as funções pertencentes a ele. Assim que todos os *commits* foram preenchidos, o último passo foi percorrer esse arquivo e para cada função, buscar no *dataset* o *sample* referente a ela e atualizá-lo como vulnerável.

3.4.2 Análise do Log de *Commits*

O segundo método de definição das funções vulneráveis é dado pela busca de padrões de texto nas mensagens dos *commits* e *pull requests*. Esses padrões, associados às vulnerabilidades, foram definidos baseando-se no trabalho de Bosu (Bosu, 2014) e estão descritos na Tabela 2. Além destes, foram definidos alguns padrões baseados nas categorias de *CWE* (CWE, 2021). Esses padrões foram selecionados por meio das categorias que o *NVD* considera em seu mecanismo de classificação e são ilustrados na Tabela 3.

Definidos os padrões, as mensagens dos *commits* e *pull requests* de cada um dos projetos foram salvas em um arquivo. O próximo passo da estratégia foi, ao fim da geração desse arquivo, percorrê-lo em busca de algum dos padrões presentes nas Tabelas 2 e 3.

Tabela 2 – Padrões - Vulnerabilidades

Tipo de vulnerabilidade	**Padrão**
Buffer Overflow	buffer overflow, stack overflow, overflow
Integer Overflow	signedness, widthness, underflow, overflow
Cross Site Scripting	cross site, CSS, XSS, htmlspecialchar
SQL Injection	SQLI, injection
Race Condition / Deadlock	race, racy, deadlock
Improper Access Control	improper access, unauthenticated, gain access, permission
Denial of Service / Crash	denial service, denial of service, race
Cross Site Request Forgery	cross site, request forgery, CSRF, XSRF
Common	vulnerability, vulnerabilities, vulnerable, vulnerables, security, insecure, hole, exploit, attack, leak, leaks, fatal, bypass, backdoor, crash, threat, expose, breach, violate, blacklist, overrun, weaknesses

Tabela 3 – Padrões - Vulnerabilidades (CWE)

Tipo de vulnerabilidade	**Padrão**
CWE	20, 22, 59, 74, 77, 78, 79, 88, 89, 91, 94, 116, 119, 120, 125, 129, 131, 134, 178, 190, 191, 193, 200, 203, 209, 212, 252, 269, 273, 276, 281, 287, 290, 294, 295, 306, 307, 311, 312, 319, 326, 327, 330, 331, 335, 338, 345, 346, 347, 352, 354, 362, 367, 369, 384, 400, 401, 404, 415, 416, 425, 426, 427, 428, 434, 436, 444, 459, 470, 476, 494, 502, 521, 522, 532, 552, 565, 601, 610, 611, 613, 617, 639, 640, 662, 665, 667, 668, 669, 670, 672, 674, 681, 682, 697, 704, 706, 732, 754, 755, 763, 770, 772, 776, 787, 798, 824, 829, 834, 835, 838, 843, 862, 863, 908, 909, 913, 915, 916, 917, 918, 920, 922, 924, 1021, 1188, 1236

Uma observação é que os padrões definidos nesta segunda tabela foram concatenados ao prefixo *"cwe-"*.

Caso algum *commit* ou *pull request* fosse considerado vulnerável, ou seja, algum dos padrões existisse em sua mensagem, todas as funções presentes nele também seriam consideradas vulneráveis. Para facilitar essa tarefa, as funções foram obtidas por meio dos dados armazenados na subseção de coleta de histórico (3.3.2), mais especificamente, por meio do campo "*commits*". Resumidamente, se dada função possuísse nos valores desse campo um *commit* dito vulnerável, ela também seria considerada vulnerável e consequentemente, o *sample* responsável por ela no *dataset*.

3.5 Preparação das bases

Essa seção é responsável por detalhar os passos da preparação das bases. Esses passos são divididos em 3 subetapas: 1) Pré-processamento; 2) Balanceamento do modelo; 3) Hiperparametrização.

3.5.1 Pré-processamento

O grande problema que deve ser resolvido nesta etapa é a presença de valores ausentes, o que dificulta a análise, já que grande parte das técnicas constroem seus modelos usando apenas os casos que possuem um conjunto completo de valores de dados (Mundfrom; Whitcomb, 1998).

Tendo como premissa manter os *samples* da base de dados mais próximos da originalidade, ao invés de realizar uma estratégia de imputação aos valores ausentes, foi decidido que somente os *samples* que possuem todas as 4 categorias de *features* (complexidade, histórico, desenvolvedor e vulnerabilidade) seriam mantidos. O quantitativo de cada um dos projetos após essa tratativa pode ser vista na Tabela 4.

Tabela 4 – Base de dados - Número de *samples*

Sistema	**Base Original**	**Base Final**
apache/httpd	27.519	6.031
cherokee/webserver	3.292	1.757
curl/curl	5.879	2.196
GNOME/libxml2	8.083	2.810
irssi/irssi	5.393	3.195
libav/libav	23.983	10.095
libexpat/libexpat	1.356	464
libssh2/libssh2	1.079	436
lighttpd/lighttpd1.4	3.258	1.623
openssl/openssl	26.978	11.885
torvalds/linux	1.056.822	449.094

Muitos algoritmos de *Machine Learning* têm melhor desempenho quando a distribuição de variáveis é gaussiana (Brownlee, 2020). Então, para finalizar o pré-processamento, foi realizada a *Quantile Transform*, que transforma variáveis numéricas de entrada ou saída para ter uma distribuição de probabilidade gaussiana (Brownlee, 2020).

3.5.2 Balanceamento

Um problema frequentemente encontrado em análises de *software* do mundo real, é que estes contém apenas algumas funções vulneráveis (casos positivos) e um grande número de não vulneráveis (casos negativos). Como consequência, os algoritmos de *Machine Learning* tradicionais têm dificuldade em lidar com bases de dados desbalanceadas

e assim, apresentam um baixo desempenho (Song, 2019). Nesta subseção, são detalhados os passos utilizados no balanceamento da base de dados.

Primeiramente, foi realizada uma análise para se obter a razão do número de funções vulneráveis e não vulneráveis. Essa etapa consistiu em agrupar todos os projetos em uma base de dados única e assim, calcular esse valor. Como resultado, foi obtido um valor aproximado de 1 função vulnerável para 3 não vulneráveis. Então, para não se perder as características da base original, foi decidido que os projetos não deveriam ser totalmente balanceados, mas sim ter essa proporção de funções vulneráveis.

Para a realização do balanceamento, foi utilizado o *SMOTE* (Chawla, 2002), uma abordagem de *oversampling* para tratamento de dados desbalanceados. O *SMOTE* utiliza cada amostra da classe minoritária e cria um número previamente definido de novas amostras sintéticas próximas a ela. Essas amostras são criadas utilizando um algoritmo *K-Nearest Neighbor*, onde são escolhidos dados aleatórios da classe minoritária e, em seguida, são definidos k-vizinhos mais próximos desses dados. Os dados sintéticos são criados entre os dados aleatórios e um dos k-vizinhos, selecionado aleatoriamente. Esse processo se repete até que a proporção de dados entre as classes seja equivalente, ou tenha alcançado um valor predefinido. Essa estratégia de balanceamento foi realizada em tempo de execução e será citada nas seções a seguir.

3.5.3 Hiperparametrização

Essa subseção é responsável por detalhar a etapa de hiperparametrização dos modelos. Foi utilizado o algoritmo de pesquisa aleatória (RandomizedSearchCV) fornecido pelo scikit-learn (Scikit-learn, 2021). Para a execução do algoritmo, o número de iterações foi definido como 100 e o número de *cross-fold-validation* foi definido como 10.

Essa etapa consistiu em concatenar as bases de dados de todos os projetos e então, criar um modelo com essa base. Esse modelo foi utilizado pelo algoritmo de pesquisa aleatória com o objetivo de obter os melhores parâmetros para cada um dos algoritmos de *Machine Learning*. Esses parâmetros são descritos a seguir:

- Random Forest: *n_estimators* representa o número de árvores; *min_samples_split* representa o número mínimo de instâncias necessárias para dividir um nó interno; *min_samples_leaf* representa o número mínimo de amostras necessárias para estar em um nó folha; *max_features* representa o número de recursos a serem considerados ao procurar a melhor divisão; *max_depth* representa a profundidade máxima da árvore; *criterion* representa a função para medir a qualidade de uma divisão; *bootstrap* indica se todas as instâncias de treinamento ou exemplos de *bootstrap* são usados para construir cada árvore (Aniche, 2020);

- Árvore de Decisão (J48): *max_features*, *max_depth*, *criterion*, *min_samples_split* e *min_samples_leaf*, ambos descritos pelo algoritmo Random Forest;
- Gradient Boosting: *loss* representa a função de perda a ser otimizada; *learning_rate* representa a taxa de aprendizagem; *n_estimators* representa o número de estágios de *boosting* a serem executados; *min_samples_split* representa o número mínimo de amostras necessárias para dividir um nó interno; *min_samples_leaf* representa o número mínimo de amostras necessárias para estar em um nó folha; *max_depth* representa a profundidade máxima dos estimadores de regressão individuais; *max_features* representa o número de recursos a serem considerados ao procurar a melhor divisão: *criterion* representa a função para medir a qualidade de uma divisão; *subsample* representa a fração de amostras a ser utilizada para se ajustar os aprendizes individuais;
- Multilayer Perceptron: *hidden_layer_sizes* o i-ésimo elemento representa o número de neurônios na i-ésima camada oculta; *activation* representa a função de ativação para a camada oculta; *solver* representa o solucionador para otimização de peso; *alpha* representa o parâmetro de penalidade L2 (termo de regularização); *learning_rate* representa a taxa de aprendizagem para atualizações de peso; *max_iter* representa o número máximo de iterações; e
- Naive Bayes: Não foi realizada a hiperparametrização.

3.6 Condução dos estudos

Essa seção descreve as etapas de condução dos três estudos realizados. A primeira subseção é responsável pelo estudo de identificação das *features* mais relevantes, seguida pela subseção que trata sobre a transferência de conhecimento entre os projetos e por fim, a subseção responsável pela predição de variantes vulneráveis.

3.6.1 Condução do estudo 1: Identificação das *features* mais relevantes

Nessa subseção, são descritos os passos executados para cada um dos 11 projetos, com o objetivo de responder à primeira questão de pesquisa (*RQ1*).

Primeiramente, a partir da base de dados pré-processada e desbalanceada, foram criadas quatro novas bases contendo as seguintes *features*: 1) *features* de complexidade; 2) *features* de complexidade + *features* de vulnerabilidade; 3) *features* de complexidade + *features* de vulnerabilidade + *features* de histórico; 4) *features* de complexidade + *features* de vulnerabilidade + *features* de histórico + *features* de desenvolvedor. A razão da base de dados estar desbalanceada é que o balanceamento foi realizado em tempo de execução e somente na base de treinamento, mantendo a base de testes livre de amostras sintéticas.

Para cada uma das quatro bases geradas, foi realizado o *cross-validation*. *Cross-validation* é um método estatístico de avaliação do desempenho de generalização que é mais estável e completo do que usar uma divisão em um treinamento e um conjunto de teste (Müller; Guido, 2016). No *cross-validation*, a base de dados é dividida repetidamente e vários modelos são treinados. A versão mais comumente usada de *cross-validation* é a *k-fold*, onde k representa o número de partes que a base será dividida. A primeira dessas partes é utilizada para testar o modelo, e as partes restantes são utilizadas para treiná-lo.

A cada iteração do *cross-validation*, foi realizado o balanceamento das partes utilizadas no treinamento do modelo. A técnica realizada no balanceamento é descrita na subseção 3.5.2. Com a base balanceada, foram treinados os modelos utilizando os algoritmos *Random Forest, Árvore de Decisão (J48), Gradient Boosting, Multilayer Perceptron e Naive Bayes*. Esses modelos foram utilizados para predizer, a cada iteração do *cross-validation*, a parte da base de dados que foi selecionada para o teste.

3.6.2 Condução do estudo 2: Transferência do conhecimento entre projetos

Para responder à segunda questão de pesquisa (*RQ2*), foi realizado o *cross-project-validation*. Essa estratégia consiste em, ao invés de dividir a base de um dos projetos em treinamento e teste, N-1 projetos são utilizados para treinar o modelo e 1 projeto é utilizado para testá-lo.

Para expandir o cenário de testes, foram realizadas 3 execuções, cada uma delas utilizando uma estratégia diferente para a criação do modelo:

- *Cross-project-validation* completo: Essa estratégia de geração do modelo consistiu em iterar sobre as 11 bases de dados pré-processadas e desbalanceadas, referentes a cada um dos projetos. A cada iteração, o projeto iterado foi utilizado para teste, e os outros 10 projetos foram concatenados e utilizados para treinamento do modelo. Essa estratégia tem como objetivo verificar se o comportamento é aceitável ao se utilizar um modelo treinado utilizando dados de projetos diferentes do analisado;
- *Cross-project-validation* parcial: Essa estratégia tem o comportamento similar ao da estratégia de geração completa, porém, por possuir um tamanho significativamente maior que os demais, o projeto torvalds/linux não foi utilizado. O objetivo dessa estratégia é verificar qual a influência de um projeto de grande porte nos resultados;
- Modelo Kernel Linux: Nesta estratégia, somente um projeto foi utilizado no treinamento do modelo e os outros 10 projetos foram utilizados para teste. O projeto escolhido para treinamento foi o torvalds/linux. Essa estratégia tem como objetivo verificar se um modelo criado utilizando somente um projeto de grande porte é suficiente para se obter bons resultados.

O próximo passo foi a realização do balanceamento do modelo gerado, utilizando os passos descritos na subseção 3.5.2. Por fim, o modelo balanceado foi utilizado na predição do projeto de teste e então, foi realizada a avaliação desse modelo.

3.6.3 Otimização da métrica *Youden Index* e Definição das métricas de análise

Após a realização das execuções das duas etapas anteriores, os modelos gerados foram submetidos a um processo visando encontrar o melhor valor da métrica *Youden Index*. Esse processo foi realizado tomando o *Score* gerado pelos modelos para cada um dos projetos. *Score* representa a probabilidade de determinado *sample* ser classificado como vulnerável ou não. Ao encontrar a otimização do *Youden Index*, esse valor foi utilizado como "corte"para o *Score*, reclassificando os *samples*, isto é, predições cujo *Score* fosse maior ou menor que esse valor, foram reclassificadas como vulnerável ou não vulnerável, respectivamente.

Para a realização das análises, foram utilizadas as métricas: 1) *Youden Index*; 2) TPR; 3) FPR. Essa decisão é reflexo do objetivo em maximizar a efetividade e minimizar o esforço nos testes, onde a métrica TPR é responsável por avaliar a efetividade e a métrica FPR, o esforço. Já a métrica *Youden Index* foi utilizada para avaliar a qualidade dos modelos. A seção 2.4 é responsável pela definição de cada uma dessas métricas.

3.6.4 Condução do estudo 3: Predição de variantes vulneráveis

Esta subseção é responsável por detalhar os passos utilizados na geração das variantes. Essa subseção é diretamente dependente das etapas anteriores e pode ser dividida em duas subetapas: 1) Identificação das funções com opções de configuração; 2) Estratégia de execução.

3.6.4.1 Identificação das funções com opções de configuração

A geração das variantes foi realizada por meio de uma estratégia de *parse* em cada um dos arquivos dos projetos, buscando por expressões que representem diretivas de compilação condicional, tais como: #ifdef, #ifndef, #if e #elif. Essas diretivas foram relacionadas às funções e arquivos aos quais elas pertencem, juntamente à expressão condicional. Uma base foi criada para cada projeto, com o intuito de salvar todas as expressões condicionais. A estrutura dessa base é ilustrada na Listagem 3.6.

Listing 3.6 – Estrutura - Base de expressões condicionais

```
{
    "file_name": {
        "function_name": [
```

```
            "feature_expression_1",
            "feature_expression_2"
            ...
        ]
        ...
    }
    ...
}
```

Uma observação importante é que as funções podem ter mais de uma estrutura condicional, isso explica a lista dessas estruturas presente em cada função. Essas expressões foram então transformadas, substituindo as opções de configuração pelos ids referentes a cada uma delas, de modo a simplificá-las. Além disso, as listas de expressões condicionais de cada função foram reduzidas a uma só expressão, concatenando-as com a condicional AND. Ao fim dessas tratativas, as expressões condicionais possuem apenas os seguintes valores:

- ID: Número utilizado para identificar a opção de configuração;
- AND, OR e NOT: Operadores *booleanos*.

Um exemplo de expressão condicional após as transformações dessa etapa possui uma estrutura similar à seguinte: 1 OR 2 AND NOT 3. Essa expressão seria satisfeita por variantes que possuam a opção de configuração 1 ou 2, desde que não possuam a opção de configuração 3. Além desta, outra base também é criada, contendo as opções de configuração e seus respectivos ids. Essas opções são obtidas através de cada uma das estruturas condicionais e os ids referentes a elas são gerados sequencialmente. Essas duas bases representam a base das expressões condicionais e opções de configuração de cada um dos projetos.

A geração de variantes foi realizada tomando as funções apontadas como vulneráveis pelas estratégias descritas nas seções anteriores e então, buscando na base de expressões condicionais aquelas pertencentes a cada uma dessas funções. Vale ressaltar que cada modelo gerado anteriormente foi responsável por um resultado diferente, já que cada projeto foi executado em 7 estratégias, por 4 algoritmos. Para cada um desses resultados, foram gerados duas novas bases, a primeira contendo somente as opções de configuração e a segunda, expressões condicionais pertencentes às funções que este resultado contempla. Uma observação importante é quanto ao tamanho do número total de opções de configuração, que variou entre os diferentes resultados.

As variantes foram geradas utilizando a heurística de amostragem LSA, detalhada na Seção 2.5. Essa heurística foi desenvolvida e adaptada ao contexto da estratégia, onde cada uma das variantes gerados pelos 3 algoritmos de *sampling* foi utilizado para checar quais funções vulneráveis foram identificadas.

3.6.4.2 Estratégia de execução

Para responder à terceira questão de pesquisa (*RQ3*), além das variantes geradas por cada uma das execuções dos modelos, foi utilizada a heurística LSA para gerar variantes referentes às expressões condicionais presentes na base. Como justificativa para a utilização dessa base na geração dessas variantes, foi necessário, para fins comparativos, um conjunto de variantes gerado de forma independente de qualquer estratégia utilizada no trabalho.

A estratégia dessa etapa se resume a encontrar o número de funções verdadeiramente vulneráveis que cada conjunto de variantes consegue identificar. Para alcançar tal objetivo, as variantes foram iteradas e a cada iteração, foi comparado o conjunto de opções de configuração dessa variante com a expressão condicional de cada função vulnerável da base de dados original, verificando se ele seria capaz de satisfazê-la. Caso a resposta fosse positiva, essa função seria contabilizada.

Uma observação é que nas bases de expressões condicionais existem somente as funções apontadas como vulneráveis pelos modelos, garantindo que as funções não existentes nessa base não fossem verificadas e consequentemente, não contabilizadas. A análise dessa execução foi baseada na comparação entre os resultados obtidos pelos conjuntos de variantes recomendados pelos modelos e pelos conjuntos gerados pelas bases dos projetos. Os resultados foram comparados através do número de funções identificadas e número de variantes necessárias para identificá-las.

4 Resultados e Discussão

Nesse capítulo, são apresentados os resultados obtidos em cada um dos estudos, juntamente às discussões referentes às questões de pesquisa.

4.1 RQ1: Quais *features* devem ser utilizadas na predição de vulnerabilidades de *software*?

Como foma de contextualização, essa questão de pesquisa foi respondida buscando encontrar o grupo de *features* com maior relevância na detecção de vulnerabilidades. A relevância em questão pode ser definida através da efetividade, onde um baixo número de FNs garante que menos funções realmente vulneráveis não foram testadas, e esforço, onde um baixo número de FPs garante que menos funções não vulneráveis foram ser testadas.

Tabela 5 – *Cross Validation* - Complexidade

Métricas	HTTPd	Webserver	Curl	Libxml2	Irssi	Libav	Libexpat	Libssh2	Lighttpd	Openssl	Linux	Média	DP
P	653	84	471	401	424	593	61	106	135	1283	60292	5863,91	17214,98
N	5378	1673	1725	2409	2771	9502	403	330	1488	10602	388802	38643,91	110779,67
Random Forest													
TP	363	50	337	232	265	332	50	61	88	636	30377	2981,00	8665,04
TN	3730	1236	1081	1893	1583	6164	270	268	1049	7973	266165	26492,00	75828,29
FP	1648	437	644	516	1188	3338	133	62	439	2629	122637	12151,91	34952,88
FN	290	34	134	169	159	261	11	45	47	647	29915	2882,91	8550,03
AUC	0,67	0,72	0,71	0,73	0,62	0,65	0,81	0,72	0,72	0,66	0,63	0,70	0,05
F,Score	0,27	0,18	0,46	0,40	0,28	0,16	0,41	0,53	0,27	0,28	0,28	0,32	0,11
Youden	0,25	0,33	0,34	0,36	0,20	0,21	0,49	0,39	0,36	0,25	0,19	0,31	0,09
TPR	0,56	0,60	0,72	0,58	0,62	0,56	0,82	0,58	0,65	0,50	0,50	0,61	0,09
FPR	0,31	0,26	0,37	0,21	0,43	0,35	0,33	0,19	0,30	0,25	0,32	0,30	0,07
Gradient Boosting													
TP	278	44	267	322	217	296	40	52	72	770	30260	2965,27	8633,64
TN	4460	1440	1244	1345	2038	7097	311	285	1042	7038	270579	26989,00	77066,19
FP	918	233	481	1064	733	2405	92	45	446	3564	118223	11654,91	33715,34
FN	375	40	204	79	207	297	21	54	63	513	30032	2898,64	8581,64
AUC	0,66	0,69	0,68	0,73	0,64	0,65	0,77	0,70	0,63	0,67	0,63	0,68	0,04
F,Score	0,30	0,24	0,44	0,36	0,32	0,18	0,41	0,51	0,22	0,27	0,29	0,32	0,10
Youden	0,26	0,38	0,29	0,36	0,25	0,25	0,43	0,35	0,23	0,26	0,20	0,30	0,07
TPR	0,43	0,52	0,57	0,80	0,51	0,50	0,66	0,49	0,53	0,60	0,50	0,56	0,10
FPR	0,17	0,14	0,28	0,44	0,26	0,25	0,23	0,14	0,30	0,34	0,30	0,26	0,09
J48													
TP	165	20	194	161	93	86	31	54	39	265	15392	1500,00	4393,66
TN	4736	1562	1405	2091	2366	8656	350	269	1334	9464	325496	32520,82	92695,63
FP	642	111	320	318	405	846	53	61	154	1138	63306	6123,09	18085,83
FN	488	64	277	240	331	507	30	52	96	1018	44900	4363,91	12821,58
AUC	0,55	0,59	0,61	0,60	0,51	0,53	0,69	0,66	0,59	0,51	0,53	0,58	0,06
F,Score	0,23	0,19	0,39	0,37	0,20	0,11	0,43	0,49	0,24	0,20	0,22	0,28	0,11
Youden	0,13	0,17	0,23	0,27	0,07	0,06	0,38	0,32	0,19	0,10	0,09	0,18	0,10
TPR	0,25	0,24	0,41	0,40	0,22	0,15	0,51	0,51	0,29	0,21	0,26	0,31	0,12
FPR	0,12	0,07	0,19	0,13	0,15	0,09	0,13	0,18	0,10	0,11	0,16	0,13	0,04
Multilayer Perceptron													
TP	421	47	316	275	255	357	43	88	73	822	30942	3058,09	8820,30
TN	3302	1381	1096	1730	1802	6264	304	192	1062	6903	278785	27529,18	79483,02
FP	2076	292	629	679	969	3238	99	138	426	3699	110017	11114,73	31298,27
FN	232	37	155	126	169	236	18	18	62	461	29350	2805,82	8394,91
AUC	0,67	0,72	0,68	0,74	0,66	0,65	0,77	0,75	0,65	0,68	0,66	0,69	0,04
F,Score	0,27	0,22	0,45	0,41	0,31	0,17	0,42	0,53	0,23	0,28	0,31	0,33	0,11
Youden	0,26	0,38	0,31	0,40	0,25	0,26	0,46	0,41	0,25	0,29	0,23	0,32	0,08
TPR	0,64	0,56	0,67	0,69	0,60	0,60	0,70	0,83	0,54	0,64	0,51	0,64	0,08
FPR	0,39	0,17	0,36	0,28	0,35	0,34	0,25	0,42	0,29	0,35	0,28	0,32	0,07
Naive Bayes													
TP	369	63	283	195	233	363	42	73	105	826	30170	2974,73	8602,51
TN	3891	1062	1222	1853	1901	6573	333	250	800	6957	263690	26230,18	75124,98
FP	1487	611	503	556	870	2929	70	80	688	3645	125112	12413,73	35655,11
FN	284	21	188	206	191	230	19	33	30	457	30122	2889,18	8612,75
AUC	0,68	0,72	0,68	0,58	0,66	0,69	0,80	0,74	0,69	0,70	0,62	0,69	0,05
F,Score	0,29	0,17	0,45	0,34	0,31	0,19	0,49	0,56	0,23	0,29	0,28	0,33	0,12
Youden	0,29	0,38	0,31	0,26	0,24	0,30	0,51	0,45	0,32	0,30	0,18	0,32	0,09
TPR	0,57	0,75	0,60	0,49	0,55	0,61	0,69	0,69	0,78	0,64	0,50	0,62	0,09
FPR	0,28	0,37	0,29	0,23	0,31	0,31	0,17	0,24	0,46	0,34	0,32	0,30	0,07

Ao todo, foram realizadas 16 execuções, alternando entre os algoritmos de *Machine Learning* e as categorias de bases. A primeira análise foi realizada com os resultados

obtidos das execuções utilizando o subconjunto de *features* de complexidade para cada um dos 4 algoritmos.

A Tabela 5 é responsável pela exibição dos resultados detalhados desta primeira análise. Uma característica importante dos resultados obtidos nessa primeira análise é quanto à diferença entre as métricas *TPR* e *FPR*. Essa primeira apresentou melhores valores, reforçando o objetivo de reduzir o número de FNs. Por outro lado, *FPR* apresentou um valor médio baixo para todos os projetos, indicando um esforço relativamente baixo em possíveis testes.

Outra observação interessante sobre os resultados é que o algoritmo *J48* foi o que melhor se comportou na redução do esforço, seguido pelos algoritmos *Gradient Boosting*, *Random Forest* e *Naive Bayes*. Já o *Multilayer Perceptron* não obteve a melhor *FPR* para nenhum dos modelos. Por outro lado, *J48* foi o algoritmo que pior se comportou na redução de FNs, não obtendo o melhor *TPR* para nenhum dos projetos. O algoritmo *Multilayer Perceptron* obteve o melhor resultado para 4 projetos e com um *TPR* médio de 0,64, foi o algoritmo com maior eficiência.

Quanto à eficiência dos projetos, libexpat/libexpat obteve o maior valor da métrica *TPR* para o algoritmo *Random Forest*, GNOME/libxml2 para o algoritmo *Gradient Boosting* e libssh2/libssh2 para os algoritmos *J48* e *Multilayer Perceptron*. Já o algoritmo *Naive Bayes* obteve o maior *TPR* para o projeto lighttpd/lighttpd1.4. Quanto à métrica *FPR*, o projeto libssh2/libssh2 obteve também o menor valor para os algoritmos *Random Forest* e *Gradient Boosting*. Os algoritmos *J48* e *Multilayer Perceptron* apresentaram um menor valor dessa métrica para o projeto cherokee/webserver. Já o *Naive Bayes* obteve um menor esforço para o projeto libexpat/libexpat.

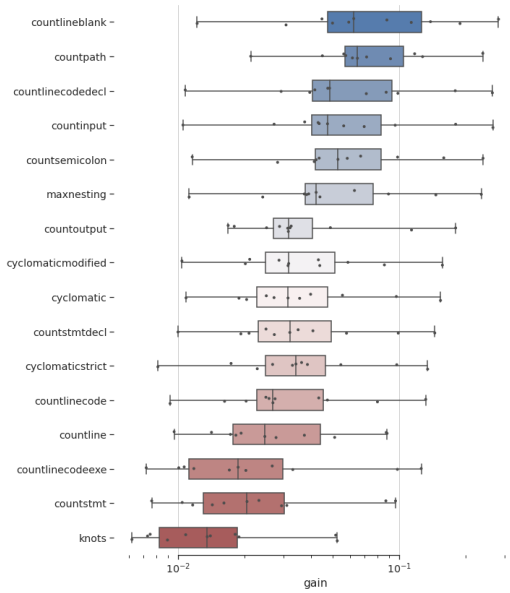

Figura 4 – Info Gain - *features* Complexidade

O projeto libexpat/libexpat obteve o melhor resultado dentre todos os analisados. Essa afirmativa pode ser validada observando o resultado obtido da métrica *Youden*, onde somente este projeto alcançou valores próximos a 0,50 para todos os algoritmos. Já o projeto torvalds/linux apresentou o pior resultado desta mesma métrica. Uma característica que diferencia este projeto em relação aos outros é o volume de dados.

Para avaliar as *features* com maior influência no modelo, foi utilizada a técnica de ganho de informação. Essa técnica é definida como a quantidade de informação fornecida pelas *features* para a categorização (Lei, 2012). A Figura 4 exibe o ganho de informação de cada uma das *features*. Dentre as *features* utilizadas, a que teve maior influência no resultado foi CountLineBlank. Por outro lado, a menor influência foi da *feature* Knots.

A baixa influência para o resultado da *feature* Knots pode estar relacionada ao baixo índice de valores distintos para ela, já que essa *feature* possui uma grande homogeneidade em seus valores. Outra característica relevante entre as *features* de complexidade é um grande número de valores zerados, fator esse que pode ter tido uma influência considerável nos modelos. Uma observação é que as *features* com maior ganho de informação (CountLineBlank, CountPath e CountLineCodeDecl) possuem poucos valores zerados e um alto índice de valores distintos.

Tabela 6 – *Cross Validation* - Complexidade + Vulnerabilidade

Métricas	HTTPd	Webserver	Curl	Libxml2	Irssi	Libav	Libexpat	Libssh2	Lighttpd	Openssl	Linux	Média	DP
P	653	84	471	401	424	593	61	106	135	1283	60292	5863,91	17214,98
N	5378	1673	1725	2409	2771	9502	403	330	1488	10602	388802	38643,91	110779,67
Random Forest													
TP	376	57	339	270	207	332	37	60	77	984	35040	3434,45	9997,80
TN	3768	1177	1088	1833	2027	6817	346	280	1226	5192	263871	26147,73	75200,66
FP	1610	496	637	576	744	2685	57	50	262	5410	124931	12496,18	35586,94
FN	277	27	132	131	217	261	24	46	58	299	25252	2429,45	7217,79
AUC	0,68	0,76	0,72	0,77	0,65	0,68	0,79	0,72	0,74	0,68	0,67	0,71	0,05
F,Score	0,28	0,18	0,47	0,43	0,30	0,18	0,48	0,56	0,32	0,26	0,32	0,34	0,12
Youden	0,28	0,38	0,35	0,43	0,22	0,28	0,47	0,41	0,39	0,26	0,26	0,34	0,08
TPR	0,58	0,68	0,72	0,67	0,49	0,56	0,61	0,57	0,57	0,77	0,58	0,62	0,08
FPR	0,30	0,30	0,37	0,24	0,27	0,28	0,14	0,15	0,18	0,51	0,32	0,28	0,10
Gradient Boosting													
TP	368	62	301	353	238	342	53	58	81	750	30229	2985,00	8617,55
TN	3624	987	1166	1140	1841	6133	235	269	1088	7260	266061	26345,82	75837,82
FP	1754	686	559	1269	930	3369	168	61	400	3342	122741	12298,09	34942,45
FN	285	22	170	48	186	251	8	48	54	533	30063	2878,91	8597,65
AUC	0,65	0,70	0,70	0,74	0,63	0,63	0,77	0,70	0,68	0,67	0,62	0,68	0,04
F,Score	0,27	0,15	0,45	0,35	0,30	0,16	0,38	0,52	0,26	0,28	0,28	0,31	0,11
Youden	0,24	0,33	0,32	0,35	0,23	0,22	0,45	0,36	0,33	0,27	0,19	0,30	0,07
TPR	0,56	0,74	0,64	0,88	0,56	0,58	0,87	0,55	0,60	0,58	0,50	0,64	0,12
FPR	0,33	0,41	0,32	0,53	0,34	0,35	0,42	0,18	0,27	0,32	0,32	0,34	0,08
J48													
TP	160	20	169	186	103	112	28	56	38	330	15206	1491,64	4337,72
TN	4660	1551	1411	2045	2322	8619	350	274	1348	9276	329319	32834,09	93803,56
FP	718	122	314	364	449	883	53	56	140	1326	59483	5809,82	16977,15
FN	493	64	302	215	321	481	33	50	97	953	45086	4372,27	12877,40
AUC	0,55	0,58	0,58	0,63	0,53	0,55	0,66	0,68	0,59	0,55	0,53	0,59	0,05
F,Score	0,21	0,18	0,35	0,39	0,21	0,14	0,39	0,51	0,24	0,22	0,23	0,28	0,11
Youden	0,11	0,17	0,18	0,31	0,08	0,10	0,33	0,36	0,19	0,13	0,10	0,19	0,10
TPR	0,25	0,24	0,36	0,46	0,24	0,19	0,46	0,53	0,28	0,26	0,25	0,32	0,11
FPR	0,13	0,07	0,18	0,15	0,16	0,09	0,13	0,17	0,09	0,13	0,15	0,13	0,03
Multilayer Perceptron													
TP	261	61	308	238	213	341	41	53	90	638	34332	3325,09	9806,64
TN	4179	1064	1239	1869	1904	5967	286	297	1003	7600	255951	25578,09	72886,01
FP	1199	609	486	540	867	3535	117	33	485	3002	132851	13065,82	37895,03
FN	392	23	163	163	211	252	20	53	45	645	25960	2538,82	7408,57
AUC	0,60	0,70	0,71	0,74	0,62	0,63	0,75	0,75	0,70	0,63	0,66	0,68	0,05
F,Score	0,25	0,16	0,49	0,40	0,28	0,15	0,37	0,55	0,25	0,26	0,30	0,32	0,12
Youden	0,18	0,36	0,37	0,37	0,19	0,20	0,38	0,40	0,34	0,21	0,23	0,29	0,09
TPR	0,40	0,73	0,65	0,59	0,50	0,58	0,67	0,50	0,67	0,50	0,57	0,58	0,09
FPR	0,22	0,36	0,28	0,22	0,31	0,37	0,29	0,10	0,33	0,28	0,34	0,28	0,07
Naive Bayes													
TP	325	67	298	206	215	355	38	63	84	770	28388	2800,82	8093,81
TN	4075	935	1205	1991	1960	6652	347	271	1033	7371	271147	26998,82	77241,64
FP	1303	738	520	418	811	2850	56	59	455	3231	117655	11645,09	33538,49
FN	328	17	173	195	209	238	23	43	51	513	31904	3063,09	9121,41
AUC	0,66	0,68	0,68	0,69	0,64	0,68	0,77	0,73	0,69	0,68	0,61	0,68	0,04
F,Score	0,28	0,15	0,46	0,40	0,30	0,19	0,49	0,55	0,25	0,29	0,28	0,33	0,12
Youden	0,26	0,36	0,33	0,34	0,21	0,30	0,48	0,42	0,32	0,30	0,17	0,32	0,08
TPR	0,50	0,80	0,63	0,51	0,51	0,60	0,62	0,59	0,62	0,60	0,47	0,59	0,09
FPR	0,24	0,44	0,30	0,17	0,29	0,30	0,14	0,18	0,31	0,30	0,30	0,27	0,08

A próxima análise foi realizada utilizando o subconjunto de *features* de complexidade somados ao subconjunto de *features* de vulnerabilidade, com o objetivo de observar se ocorre alguma diferença significante em relação à efetividade e esforço. O resultado da execução que foi utilizado nesta análise pode ser visualizado na Tabela 6.

Pode-se notar uma pequena melhoria na média dos resultados da métrica *TPR* para os algoritmos *Random Forest*, *Gradient Boosting* e *J48*, indicando uma maior eficiência em relação à primeira análise. Por outro lado, os algoritmos *Multilayer Perceptron* e *Naive Bayes* sofreram leves reduções dessa mesma métrica (0,64 → 0,58 e 0,62 → 0,59, respectivamente). A redução de esforço também obteve uma melhoria, onde *Random Forest*, *Multilayer Perceptron* e *Naive Bayes* obtiveram uma redução e *Gradient Boosting* um aumento da métrica *FPR*. Já o algoritmo *J48* manteve o valor dessa métrica.

O algoritmo que obteve o melhor resultado quanto à eficiência passou a ser o *Gradient Boosting*, obtendo uma média de 0,64 para a métrica *TPR*. Já em relação à redução do esforço, *J48* continua sendo o melhor algoritmo neste cenário, com um valor médio da métrica *FPR* de 0,13.

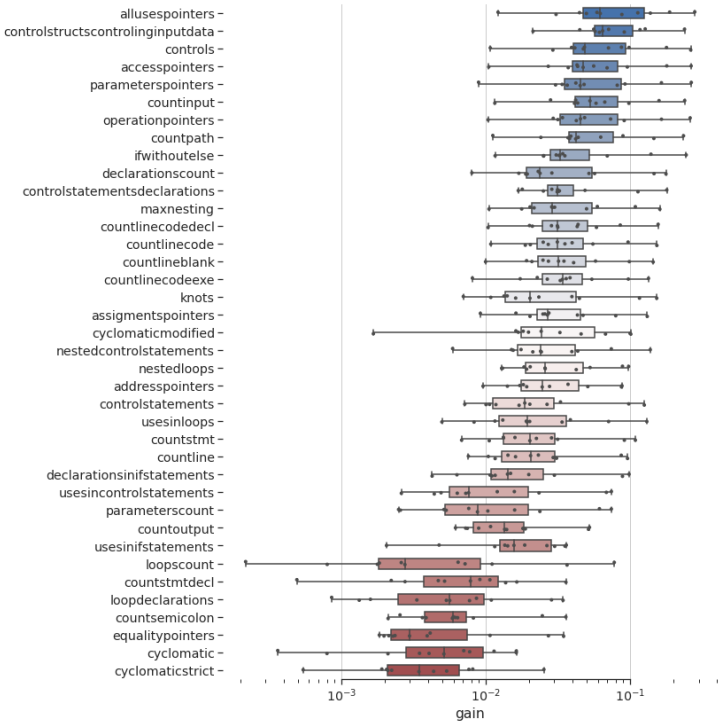

Figura 5 – Info Gain - *Features* Complexidade + Vulnerabilidade

Quanto aos modelos, notou-se uma relação entre o resultado e o algoritmo utilizado. Utilizando os algoritmos *Multilayer Perceptron* e *Naive Bayes*, cherokee/webserver obteve a maior eficiência dentre todos os projetos, já os algoritmos *Random Forest*, *Gradient Boosting* e *J48* obtiveram suas maiores eficiências com os projetos openssl/openssl, GNOME/libxml2 e libssh2/libssh2, respectivamente. A redução de esforço também foi

bastante relativa, obtendo o melhor resultado para o projeto cherokee/webserver utilizando o algoritmo *J48* (0,07).

O projeto libexpat/libexpat continuou sendo o melhor dentre os analisados, já o pior comportamento foi obtido pelo projeto irssi/irssi, ambos considerando a métrica *Youden*. Uma diferença entre esses dois projetos é quanto à quantidade de valores distintos para as métricas, sendo este primeiro responsável por uma alta incidência dessas particularidades e o segundo, uma baixa incidência.

Quanto à análise das *features* (Figura 5), AllUsesPointers foi a responsável pelo maior ganho de informação, seguida pelas *features* ControlStructsControlingInputData e Controls. Já CyclomaticStrict, Cyclomatic e EqualityPointers foram as *features* responsáveis pelo menor ganho de informação. O mais interessante é que, dentre as 10 *features* com maior influência no resultado, 8 pertencem às *features* de vulnerabilidade.

Tabela 7 – *Cross Validation* - Complexidade + Vulnerabilidade + Histórico

Métricas	HTTPd	Webserver	Curl	Libxml2	Irssi	Libav	Libexpat	Libssh2	Lighttpd	Openssl	Linux	Média	DP
P	653	84	471	401	424	593	61	106	135	1283	60292	5863,91	17214,98
N	5378	1673	1725	2409	2771	9502	403	330	1488	10602	388802	38643,91	110779,67
Random Forest													
TP	457	64	274	340	272	463	48	64	102	907	37052	3640,27	10568,46
TN	3729	1200	1483	1573	1782	5356	293	282	1112	7697	284225	28066,55	81033,85
FP	1649	473	242	836	989	4146	110	48	376	2905	104577	10577,36	29750,67
FN	196	20	197	61	152	130	13	42	33	376	23240	2223,64	6646,76
AUC	0,75	0,80	0,80	0,82	0,69	0,74	0,83	0,77	0,83	0,79	0,72	0,78	0,04
F,Score	0,33	0,21	0,56	0,43	0,32	0,18	0,44	0,59	0,33	0,36	0,37	0,37	0,12
Youden	0,39	0,48	0,44	0,50	0,28	0,34	0,51	0,46	0,50	0,43	0,35	0,43	0,07
TPR	0,70	0,76	0,58	0,85	0,64	0,78	0,79	0,60	0,76	0,71	0,61	0,71	0,08
FPR	0,31	0,28	0,14	0,35	0,36	0,44	0,27	0,15	0,25	0,27	0,27	0,28	0,08
Gradient Boosting													
TP	432	70	308	288	259	339	47	61	110	867	33266	3277,00	9485,98
TN	3387	870	1308	1697	1733	6615	303	262	1145	7421	291348	28735,36	83077,65
FP	1991	803	417	712	1038	2887	100	68	343	3181	97454	9908,55	27703,40
FN	221	14	163	113	165	254	14	45	25	416	27026	2586,91	7729,21
AUC	0,68	0,72	0,75	0,74	0,65	0,67	0,80	0,70	0,82	0,74	0,69	0,72	0,05
F,Score	0,28	0,15	0,52	0,41	0,30	0,18	0,45	0,52	0,37	0,33	0,35	0,35	0,12
Youden	0,29	0,35	0,41	0,42	0,24	0,27	0,52	0,37	0,58	0,38	0,30	0,38	0,10
TPR	0,66	0,83	0,65	0,72	0,61	0,57	0,77	0,58	0,81	0,68	0,55	0,68	0,09
FPR	0,37	0,48	0,24	0,30	0,37	0,30	0,25	0,21	0,23	0,30	0,25	0,30	0,08
J48													
TP	196	19	220	191	103	132	29	48	61	480	18495	1815,82	5275,89
TN	4558	1529	1385	2032	2360	8486	363	237	1366	9138	329977	32857,36	94002,64
FP	820	144	340	377	411	1016	40	93	122	1464	58825	5786,55	16777,63
FN	457	65	251	210	321	461	32	58	74	803	41797	4048,09	11939,32
AUC	0,57	0,57	0,63	0,66	0,55	0,56	0,69	0,59	0,68	0,62	0,56	0,61	0,05
F,Score	0,23	0,15	0,43	0,39	0,22	0,15	0,45	0,39	0,38	0,30	0,27	0,31	0,10
Youden	0,15	0,14	0,27	0,32	0,09	0,12	0,38	0,17	0,37	0,24	0,16	0,22	0,10
TPR	0,30	0,23	0,47	0,48	0,24	0,22	0,48	0,45	0,45	0,37	0,31	0,36	0,10
FPR	0,15	0,09	0,20	0,16	0,15	0,11	0,10	0,28	0,08	0,14	0,15	0,15	0,05
Multilayer Perceptron													
TP	312	57	329	279	207	355	44	62	98	792	40077	3873,82	11450,26
TN	4247	1224	1314	1870	2024	6675	321	282	1115	7581	278619	27752,00	79366,21
FP	1131	449	411	539	747	2827	82	48	373	3021	110183	10891,91	31413,79
FN	341	27	142	122	217	238	17	44	37	491	20215	1990,09	5764,95
AUC	0,70	0,75	0,79	0,79	0,63	0,69	0,80	0,75	0,79	0,71	0,75	0,74	0,05
F,Score	0,30	0,19	0,54	0,46	0,30	0,19	0,47	0,57	0,32	0,31	0,38	0,37	0,12
Youden	0,27	0,41	0,46	0,47	0,22	0,30	0,52	0,44	0,48	0,33	0,38	0,39	0,09
TPR	0,48	0,68	0,70	0,70	0,49	0,60	0,72	0,58	0,73	0,62	0,66	0,63	0,08
FPR	0,21	0,27	0,24	0,22	0,27	0,30	0,20	0,15	0,25	0,28	0,28	0,24	0,04
Naive Bayes													
TP	334	54	291	204	224	344	43	62	106	910	33129	3245,55	9452,83
TN	4294	1170	1285	2092	1968	7372	326	269	939	6976	268946	26876,09	76586,35
FP	1084	503	440	317	803	2130	77	61	549	3626	119856	11767,82	34195,33
FN	319	30	180	197	200	249	18	44	29	373	27163	2618,36	7762,57
AUC	0,71	0,69	0,72	0,73	0,66	0,73	0,79	0,73	0,74	0,74	0,65	0,72	0,04
F,Score	0,32	0,17	0,48	0,44	0,31	0,22	0,48	0,54	0,27	0,31	0,31	0,35	0,11
Youden	0,31	0,34	0,36	0,38	0,24	0,36	0,51	0,40	0,42	0,37	0,24	0,36	0,07
TPR	0,51	0,64	0,62	0,51	0,53	0,58	0,70	0,58	0,79	0,71	0,55	0,61	0,09
FPR	0,20	0,30	0,26	0,13	0,29	0,22	0,19	0,18	0,37	0,34	0,31	0,25	0,07

A terceira análise foi realizada adicionando as *features* de histórico às atuais. Os resultados desta terceira análise (Tabela 7), apresentaram uma melhoria da métrica *Youden*, indicando um melhor comportamento dos modelos na detecção das funções vulneráveis.

A eficiência do modelo apresentou uma considerável melhoria em relação às análises

anteriores. Isto fica evidente quando se observa os resultados da métrica *TPR* para o algoritmo *Random Forest*, onde o menor resultado obtido foi de 0,58, para o projeto curl/curl. Outro ponto importante a ser destacado é quanto à redução do desvio padrão dessa mesma métrica em todos os algoritmos, indicando uma menor dispersão dos valores em torno da média. O esforço também foi outro ponto bastante positivo. Dentre todos os algoritmos, o valor médio da métrica *FPR* teve seu pior resultado para o algoritmo *Gradient Boosting* (0,30), sendo relativamente melhor que o pior resultado dentre todos os algoritmos da análise anterior (0,34).

Quanto aos algoritmos, *Random Forest* obteve o melhor comportamento considerando a eficiência do modelo, tendo uma melhoria em relação à média da métrica *TPR* (0,62 → 0,71). Já em relação à redução do esforço, o melhor algoritmo foi o *J48*, mesmo tendo uma leve piora em relação à análise anterior (0,13 → 0,15). Os projetos responsáveis pelo melhor e pior resultado nessa execução continuaram sendo o libexpat/libexpat e o irssi/irssi. Uma observação é que, mesmo com a inserção das *features* de histórico, dentre todos os projetos analisados, irssi/irssi continuou tendo a maior similaridade entre os valores presentes em suas *features*.

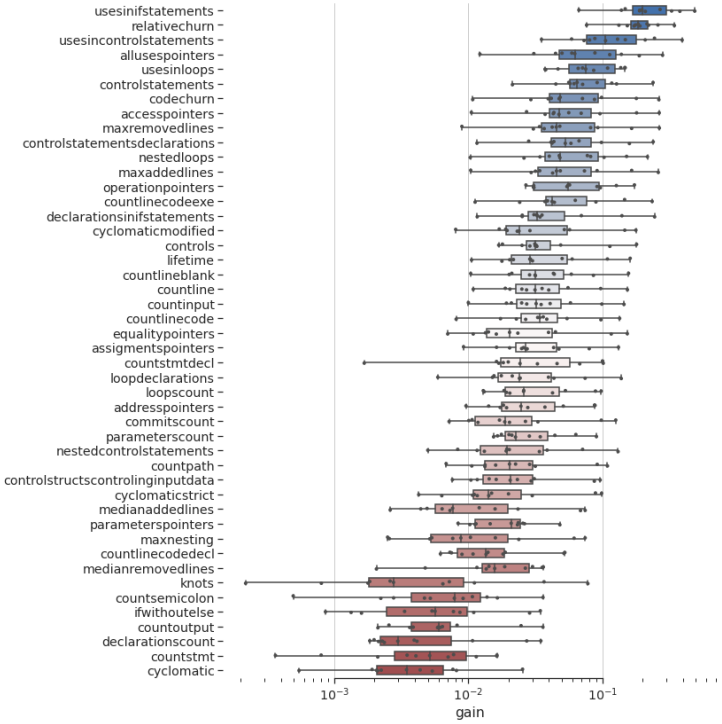

Figura 6 – Info Gain - *Features* Complexidade + Vulnerabilidade + Histórico

A Figura 6 é responsável por exibir o resultado do ganho de informação das *features* utilizadas nessa execução. A *feature* UsesInIfStatements foi responsável pelo maior ganho, seguida pelas *features* RelativeChurn, inserida nessa etapa e UsesInControlStatements. Quanto ao menor ganho, das 3 *features* mais significantes, 2 são de complexidade e 1

é de vulnerabilidade. Dentre as 10 responsáveis pelo maior e menor ganho, existem 4 *features* de histórico, sendo RelativeChurn, CodeChurn e MaxRemovedLines presentes no primeiro grupo e MedianRemovedLines presentes no segundo grupo. O restante das *features* inseridas nesta etapa foram responsáveis por um ganho de informação mediano.

Tabela 8 – *Cross Validation* - Complexidade + Vulnerabilidade + Histórico + Desenvolvedor

Métricas	HTTPd	Webserver	Curl	Libxml2	Irssi	Libav	Libexpat	Libssh2	Lighttpd	Openssl	Linux	Média	DP
P	653	84	471	401	424	593	61	106	135	1283	60292	5863,91	17214,98
N	5378	1673	1725	2409	2771	9502	403	330	1488	10602	388802	38643,91	110779,67
Random Forest													
TP	405	65	288	344	293	405	55	60	120	946	37441	3674,73	10680,59
TN	3982	1183	1440	1539	1666	6704	261	305	924	7399	282866	28024,45	80622,17
FP	1396	490	285	870	1105	2798	142	25	564	3203	105936	10619,45	30158,08
FN	248	19	183	57	131	188	6	46	15	337	22851	2189,18	6534,65
AUC	0,74	0,80	0,79	0,82	0,69	0,77	0,84	0,77	0,83	0,79	0,73	0,78	0,04
F,Score	0,33	0,20	0,55	0,43	0,32	0,21	0,43	0,63	0,29	0,35	0,37	0,37	0,12
Youden	0,36	0,48	0,45	0,50	0,29	0,39	0,55	0,49	0,51	0,44	0,35	0,44	0,08
TPR	0,62	0,77	0,61	0,86	0,69	0,68	0,90	0,57	0,89	0,74	0,62	0,72	0,11
FPR	0,26	0,29	0,17	0,36	0,40	0,29	0,35	0,08	0,38	0,30	0,27	0,29	0,09
Gradient Boosting													
TP	438	52	297	276	191	415	45	77	104	820	35537	3477,45	10140,46
TN	3301	1135	1335	1819	2104	5971	324	234	982	7871	289760	28621,45	82610,93
FP	2077	538	390	590	667	3531	79	96	506	2731	99042	10022,45	28171,74
FN	215	32	174	125	233	178	16	29	31	463	24755	2386,45	7074,66
AUC	0,69	0,67	0,75	0,75	0,63	0,70	0,83	0,75	0,76	0,75	0,71	0,73	0,05
F,Score	0,28	0,15	0,51	0,44	0,30	0,18	0,49	0,55	0,28	0,34	0,36	0,35	0,13
Youden	0,28	0,30	0,40	0,44	0,21	0,33	0,54	0,44	0,43	0,38	0,33	0,37	0,09
TPR	0,67	0,62	0,63	0,69	0,45	0,70	0,74	0,73	0,77	0,64	0,59	0,66	0,08
FPR	0,39	0,32	0,23	0,24	0,24	0,37	0,20	0,29	0,34	0,26	0,25	0,28	0,06
J48													
TP	214	24	213	184	101	144	21	53	48	487	19277	1887,82	5500,41
TN	4611	1532	1376	2032	2317	8642	333	254	1353	9120	327272	32622,00	93222,94
FP	767	141	349	377	454	860	70	76	135	1482	61530	6021,91	17557,92
FN	439	60	258	217	323	449	40	53	87	796	41015	3976,09	11714,75
AUC	0,59	0,60	0,62	0,65	0,54	0,58	0,59	0,63	0,63	0,62	0,57	0,60	0,03
F,Score	0,26	0,19	0,41	0,38	0,21	0,18	0,28	0,45	0,30	0,30	0,27	0,29	0,08
Youden	0,19	0,20	0,25	0,30	0,07	0,15	0,17	0,27	0,26	0,24	0,16	0,21	0,06
TPR	0,33	0,29	0,45	0,46	0,24	0,24	0,34	0,50	0,36	0,38	0,32	0,35	0,08
FPR	0,14	0,08	0,20	0,16	0,16	0,09	0,17	0,23	0,09	0,14	0,16	0,15	0,04
Multilayer Perceptron													
TP	394	62	279	268	235	435	51	56	74	703	38790	3758,82	11079,49
TN	3633	1144	1452	1952	1835	5399	310	301	1286	8085	279708	27736,82	79712,02
FP	1745	529	273	457	936	4103	93	29	202	2517	109094	10907,09	31072,63
FN	259	22	192	133	189	158	10	50	61	580	21502	2105,09	6135,71
AUC	0,69	0,76	0,77	0,79	0,63	0,70	0,84	0,76	0,75	0,71	0,74	0,74	0,05
F,Score	0,28	0,18	0,55	0,48	0,29	0,17	0,50	0,59	0,36	0,31	0,37	0,37	0,13
Youden	0,28	0,42	0,43	0,48	0,22	0,30	0,61	0,44	0,41	0,31	0,36	0,39	0,10
TPR	0,60	0,74	0,59	0,67	0,55	0,73	0,84	0,53	0,55	0,55	0,64	0,64	0,09
FPR	0,32	0,32	0,16	0,19	0,34	0,43	0,23	0,09	0,14	0,24	0,28	0,25	0,10
Naive Bayes													
TP	514	58	278	197	283	377	43	61	83	996	35798	3517,09	10211,57
TN	2926	1185	1346	2145	1646	7131	334	277	1163	6480	262304	26085,18	74731,07
FP	2452	488	379	264	1125	2371	69	53	325	4122	126498	12558,73	36052,06
FN	139	26	193	204	141	216	18	45	52	287	24494	2346,82	7004,06
AUC	0,72	0,71	0,73	0,73	0,67	0,75	0,79	0,74	0,76	0,76	0,68	0,73	0,03
F,Score	0,28	0,18	0,49	0,46	0,31	0,23	0,50	0,55	0,31	0,31	0,32	0,36	0,12
Youden	0,33	0,40	0,37	0,38	0,26	0,39	0,53	0,41	0,40	0,39	0,27	0,38	0,07
TPR	0,79	0,69	0,59	0,49	0,67	0,64	0,70	0,58	0,61	0,78	0,59	0,65	0,08
FPR	0,46	0,29	0,22	0,11	0,41	0,25	0,17	0,16	0,22	0,39	0,33	0,27	0,11

A última análise (Tabela 8) foi realizada adicionando o subconjunto de *features* de desenvolvedor às *features* utilizadas na terceira análise. O primeiro ponto a destacar é quanto a um pequeno aumento na média da métrica *TPR* para os algoritmos: *Random Forest* (0,71 → 0,72), *Multilayer Perceptron* (0,63 → 0,64) e *Naive Bayes* (0,61 → 0,65). Já os algoritmos *Gradient Boosting* e *J48* sofreram uma leve redução em seus valores médios (0,68 → 0,66 e 0,36 → 0,35, respectivamente). Quanto à redução de esforço, o algoritmo *J48* manteve o valor da métrica *FPR*, *Random Forest*, *Multilayer Perceptron* e *Naive Bayes* tiveram um aumento dessa métrica. O único algoritmo que apresentou uma melhoria foi o *Gradient Boosting*. Considerando a métrica *Youden*, o algoritmo *Random Forest* obteve o melhor comportamento dentre os analisados.

O projeto libexpat/libexpat obteve a melhor eficiência dentre todos os outros, utilizando os algoritmos *Random Forest* e *Multilayer Perceptron*. Por outro lado, esse projeto necessita de um esforço médio (0,35 e 0,23, respectivamente), considerando esses mesmos

algoritmos. lighttpd/lighttpd1.4 foi o projeto que apresentou a maior eficiência utilizando o algoritmo *Gradient Boosting* (0,77), libssh2/libssh2 utilizando o algoritmo *J48* (0,50) e apache/httpd utilizando *Naive Bayes* (0,50). Considerando a redução de esforço, o projeto libssh2/libssh2 obteve o melhor comportamento para os algoritmos *Random Forest*, *Multilayer Perceptron*. Já *Gradient Boosting*, *J48* e *Naive Bayes* obtiveram seus melhores valores da métrica *FPR* utilizando os projetos libexpat/libexpat, cherokee/webserver e GNOME/libxml2.

Baseando-se nos resultados da métrica *Youden*, pode-se dizer que o projeto libexpat/libexpat obteve o melhor comportamento dentre todos os analisados. Um ponto a se destacar é que este projeto obteve o melhor resultado para a métrica *Youden* em quase todos os algoritmos, superado pelo GNOME/libxml2 apenas no *J48*. Por outro lado, o projeto irssi/irssi continuou sendo o projeto com menor *Youden* médio, podendo ser considerado o pior projeto.

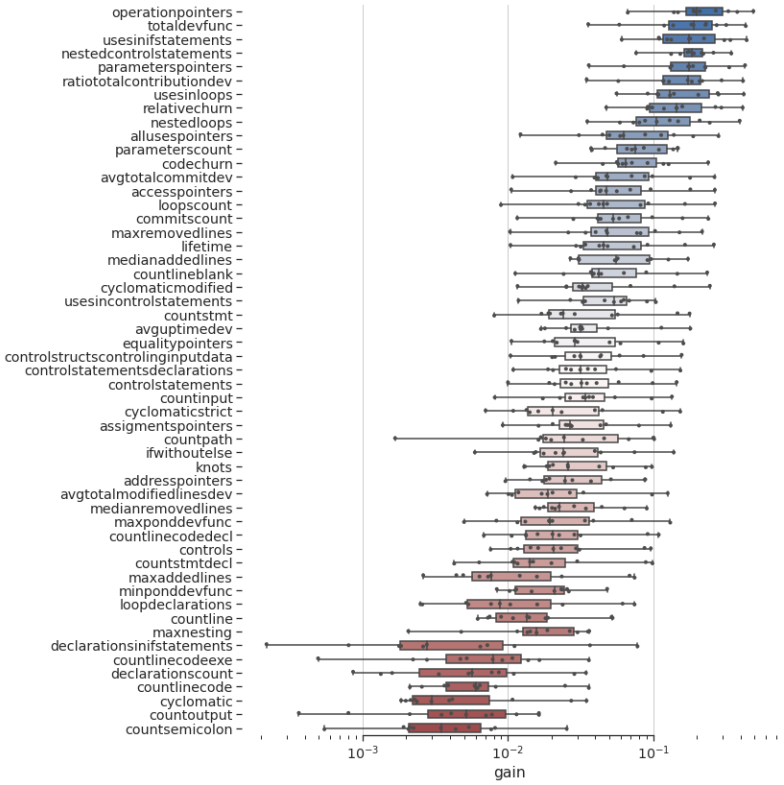

Figura 7 – Info Gain - *Features* Complexidade + Vulnerabilidade + Histórico + Desenvolvedor

O ganho de informação é exibido na Figura 7. As 3 *features* com maior ganho de informação foram OperationPointers, TotalDevFunc e UsesInIfStatements, duas delas *features* de vulnerabilidade e uma de desenvolvedor. Por outro lado, nenhuma *feature* inserida nesta etapa ficou entre as 10 com menor ganho. Com base no resultado obtido nesta última análise, fica evidente que não se obteve um ganho significante ao se inserir essas novas *features*.

4.2 RQ2: Os modelos de predição podem ser transferidos entre projetos de *software*?

Para responder essa questão de pesquisa, foi primeiramente analisado o resultado da execução do *cross-project-validation* contendo todos os projetos. Em seguida, o comportamento desta primeira execução foi comparado ao resultado do *cross-project-validation* onde o maior projeto, torvalds/linux, foi removido. Esta comparação permite observar a influência de um projeto de grande porte no modelo. Por fim, a terceira e última análise foi realizada com base no resultado da execução utilizando o modelo criado somente com o projeto torvalds/linux. A motivação desta análise é verificar se um modelo criado com um projeto grande é suficiente para se obter bons resultados.

Assim como na seção anterior, os resultados foram analisados em termos de eficiência e esforço. As métricas *TPR*, *FPR* e *Youden* foram utilizadas para realizar o comparativo das execuções. A primeira execução apresentou resultados interessantes em relação à eficiência, apresentando no pior caso um valor médio de 0,53 para a métrica *TPR*. Por outro lado, o esforço médio teve um leve aumento, obtendo para a métrica *FPR* um valor médio de 0,38, também para o pior caso. A Tabela 9 é responsável por exibir o resultado completo desta execução.

Uma consideração importante a ser mencionada é que, diferente da análise da RQ1, onde o modelo foi criado utilizando o mesmo projeto ao qual ele analisou, na análise atual, enquanto um projeto é utilizado para teste, os outros N-1 projetos restantes são utilizados na criação do modelo.

A melhor transferência de conhecimento foi obtida ao se prever o projeto libexpat/libexpat. O resultado dessa execução obteve uma média de 0,73 para a métrica *TPR*. Utilizando o algoritmo *Multilayer Perceptron*, essa execução conseguiu apontar um grande número de funções vulneráveis, obtendo um *TPR* de 0,80.

Quanto ao esforço, o projeto lighttpd/lighttpd1.4 atingiu o melhor resultado dentre os demais (*FPR* médio 0,25). Considerando a métrica *Youden* como responsável por avaliar a qualidade do modelo, pode-se dizer que o melhor comportamento foi encontrado no projeto libexpat/libexpat. O resultado para este projeto apresentou a maior média (0,47) dentre os analisados. Um maior detalhamento do resultado dessa métrica entre os projetos pode ser visto na Figura 8.

Os algoritmos *Random Forest*, *Gradient Boosting* e *Naive Bayes* foram os que melhor se comportaram em relação à eficiência. Já o *J48* obteve o pior comportamento, embora não tenha sido responsável pela maior dispersão dos resultados, quando comparado aos outros algoritmos. Em relação à redução do esforço, *Multilayer Perceptron* foi o algoritmo que apresentou o melhor resultado médio, obtendo um valor de 0,23 para

Tabela 9 – *Cross-project-validation completo* - Resultado

Métricas	HTTPd	Webserver	Curl	Libxml2	Irssi	Libav	Libexpat	Libssh2	Lighttpd	Openssl	Linux	Média	DP
P	653	84	471	401	424	593	61	106	135	1283	60292	5863,91	17214,98
N	5378	1673	1725	2409	2771	9502	403	330	1488	10602	388802	38643,91	110779,67
Random Forest													
TP	439	59	335	287	225	428	47	88	77	875	37763	3693,00	10776,36
TN	3592	1147	1215	1461	2025	6186	310	226	1227	7296	273055	27067,27	77819,78
FP	1786	526	510	948	746	3316	93	104	261	3306	115747	11576,64	32960,53
FN	214	25	136	114	199	165	14	18	58	408	22529	2170,91	6438,73
AUC	0,72	0,73	0,75	0,66	0,66	0,74	0,81	0,79	0,74	0,73	0,72	0,73	0,04
F,Score	0,31	0,18	0,51	0,35	0,32	0,20	0,47	0,59	0,33	0,32	0,35	0,36	0,12
Youden	0,34	0,39	0,42	0,32	0,26	0,37	0,54	0,52	0,39	0,37	0,33	0,39	0,08
TPR	0,67	0,70	0,71	0,72	0,53	0,72	0,77	0,83	0,57	0,68	0,63	0,68	0,08
FPR	0,33	0,31	0,30	0,39	0,27	0,35	0,23	0,32	0,18	0,31	0,30	0,30	0,06
Gradient Boosting													
TP	429	62	360	280	298	381	46	66	87	817	39410	3839,64	11250,41
TN	3955	886	1264	1893	1622	7413	262	269	1108	7979	274584	27385,00	78213,74
FP	1423	787	461	516	1149	2089	141	61	380	2623	114218	11258,91	32567,71
FN	224	22	111	121	126	212	15	40	48	466	20882	2024,27	5964,62
AUC	0,76	0,66	0,80	0,78	0,68	0,76	0,73	0,76	0,72	0,72	0,74	0,74	0,04
F,Score	0,34	0,13	0,56	0,47	0,32	0,25	0,37	0,57	0,29	0,35	0,37	0,36	0,12
Youden	0,39	0,27	0,50	0,48	0,29	0,42	0,40	0,44	0,39	0,39	0,36	0,39	0,07
TPR	0,66	0,74	0,76	0,70	0,70	0,64	0,75	0,62	0,64	0,64	0,65	0,68	0,05
FPR	0,26	0,47	0,27	0,21	0,41	0,22	0,35	0,18	0,26	0,25	0,29	0,29	0,08
J48													
TP	431	50	258	164	217	371	35	48	73	722	21570	2176,27	6136,06
TN	2699	954	1044	1438	1757	5251	280	210	1008	6610	295248	28772,64	84289,39
FP	2679	719	681	971	1014	4251	123	120	480	3992	93554	9871,27	26500,23
FN	222	34	213	237	207	222	26	58	62	561	38722	3687,64	11079,77
AUC	0,58	0,58	0,57	0,51	0,56	0,58	0,63	0,54	0,61	0,59	0,56	0,58	0,03
F,Score	0,23	0,12	0,37	0,21	0,26	0,14	0,32	0,35	0,21	0,24	0,25	0,25	0,07
Youden	0,16	0,17	0,15	0,01	0,15	0,18	0,27	0,09	0,22	0,19	0,12	0,15	0,07
TPR	0,66	0,60	0,55	0,41	0,51	0,63	0,57	0,45	0,54	0,56	0,36	0,53	0,09
FPR	0,50	0,43	0,39	0,40	0,37	0,45	0,31	0,36	0,32	0,38	0,24	0,38	0,07
Multilayer Perceptron													
TP	420	61	359	247	223	362	49	80	68	815	33306	3271,82	9500,07
TN	4018	1220	1178	1986	2096	7422	306	251	1260	8462	284225	28402,18	80941,12
FP	1360	453	547	423	675	2080	97	79	228	2140	104577	10241,73	29839,71
FN	233	23	112	154	201	231	12	26	67	468	26986	2592,09	7715,07
AUC	0,77	0,79	0,79	0,77	0,69	0,76	0,83	0,79	0,71	0,79	0,69	0,76	0,05
F,Score	0,35	0,20	0,52	0,46	0,34	0,24	0,47	0,60	0,32	0,38	0,34	0,38	0,11
Youden	0,39	0,46	0,45	0,44	0,28	0,39	0,56	0,52	0,35	0,43	0,28	0,41	0,08
TPR	0,64	0,73	0,76	0,62	0,53	0,61	0,80	0,75	0,50	0,64	0,55	0,65	0,10
FPR	0,25	0,27	0,32	0,18	0,24	0,22	0,24	0,24	0,15	0,20	0,27	0,23	0,04
Naive Bayes													
TP	413	64	256	260	283	441	47	66	92	1038	36906	3624,18	10528,08
TN	3688	1060	1318	1780	1704	5857	317	258	972	5941	259347	25658,36	73923,94
FP	1690	613	407	629	1067	3645	86	72	516	4661	129455	12985,55	36858,39
FN	240	20	215	141	141	152	14	40	43	245	23386	2239,73	6687,54
AUC	0,72	0,71	0,71	0,73	0,67	0,73	0,80	0,75	0,71	0,75	0,69	0,72	0,03
F,Score	0,30	0,17	0,45	0,40	0,32	0,19	0,48	0,54	0,25	0,30	0,33	0,34	0,11
Youden	0,32	0,40	0,31	0,39	0,28	0,36	0,56	0,40	0,33	0,37	0,28	0,36	0,07
TPR	0,63	0,76	0,54	0,65	0,67	0,74	0,77	0,62	0,68	0,81	0,61	0,68	0,08
FPR	0,31	0,37	0,24	0,26	0,39	0,38	0,21	0,22	0,35	0,44	0,33	0,32	0,07

a métrica *FPR*. Além disso, considerando o desvio padrão, nota-se que esse algoritmo apresentou a menor dispersão dos resultados dentre todos os analisados. Uma observação importante é que a dispersão dos valores das métricas *TPR* e *FPR* é consideravelmente baixa para todos os algoritmos. A Figura 9 ilustra com mais detalhamento o resultado das métricas *TPR* e *FPR*, para todos os algoritmos.

A segunda análise consiste em verificar a influência que projetos de grande porte tem no modelo. Para a realização desta etapa, foi removido o torvalds/linux, maior projeto deste estudo, com cerca de 940 mil funções. O resultado desta execução encontra-se na Tabela 10.

Analisando os resultados da métrica *Youden* para todos os projetos separadamente (Figura 10), nota-se uma menor dispersão dos valores em relação à análise anterior. O projeto libexpat/libexpat continuou apresentando o melhor resultado, embora outros projetos tenham apresentando um valor mínimo maior que ele. O pior projeto foi o irssi/irssi, responsável pela menor média *Youden* entre todos os analisados.

No geral, exceto para o algoritmo *J48*, a eficiência não sofreu grandes alterações. O

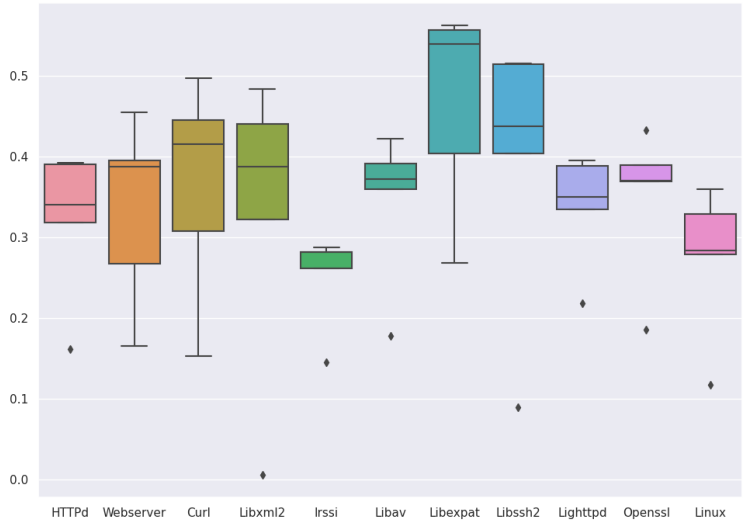

Figura 8 – Cross-project-validation completo - Métrica *Youden-Index*

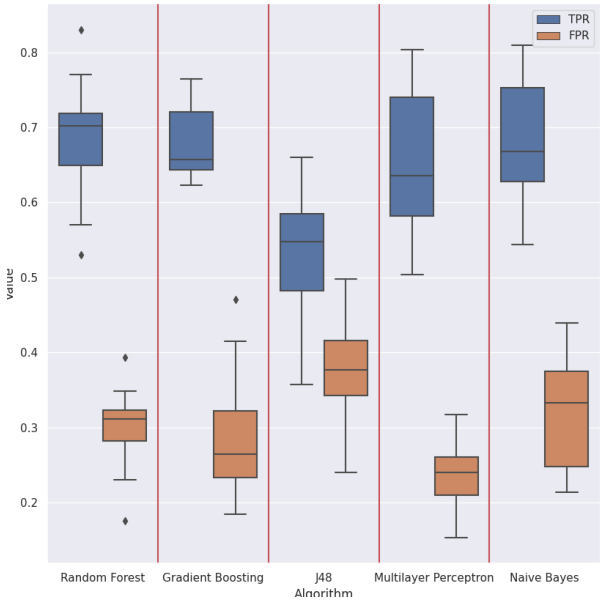

Figura 9 – Cross-project-validation completo - Métricas *TPR* e *FPR*

algoritmo *Multilayer Perceptron* apresentou uma melhoria média da métrica *TPR* (0,65 → 0,69). Já *Random Forest*, *Gradient Boosting* e *J48* apresentaram uma piora desta mesma métrica, sendo esse último responsável pela maior diferença, onde o *TPR* sofreu uma redução de 0,53 → 0,41. Já o algoritmo *Naive Bayes* não sofreu alterações. O principal ganho obtido nessa execução foi em relação à redução do esforço, onde somente o algoritmo *Multilayer Perceptron* apresentou uma piora do resultado apresentado pela métrica *FPR*. A maior dessas reduções aconteceu utilizando o algoritmo *J48* (0,38 → 0,25). Um maior

Tabela 10 – *Cross-project-validation parcial* - Resultado

Métricas	HTTPd	Webserver	Curl	Libxml2	Irssi	Libav	Libexpat	Libssh2	Lighttpd	Openssl	Média	DP
P	653	84	471	401	424	593	61	106	135	1283	421,10	354,06
N	5378	1673	1725	2409	2771	9502	403	330	1488	10602	3628,10	3488,90
Random Forest												
TP	487	52	327	292	214	407	36	77	93	880	286,50	247,58
TN	3474	1163	1238	1505	2175	6917	346	238	1088	7938	2608,20	2571,76
FP	1904	510	487	904	596	2585	57	92	400	2664	1019,90	940,41
FN	166	32	144	109	210	186	25	29	42	403	134,60	111,52
AUC	0,75	0,71	0,77	0,72	0,68	0,78	0,78	0,75	0,74	0,78	0,75	0,03
F,Score	0,32	0,16	0,51	0,37	0,35	0,23	0,47	0,56	0,30	0,36	0,36	0,12
Youden	0,39	0,31	0,41	0,35	0,29	0,41	0,45	0,45	0,42	0,43	0,39	0,05
TPR	0,75	0,62	0,69	0,73	0,50	0,69	0,59	0,73	0,69	0,69	0,67	0,07
FPR	0,35	0,30	0,28	0,38	0,22	0,27	0,14	0,28	0,27	0,25	0,27	0,06
Gradient Boosting												
TP	417	66	276	239	263	478	37	64	67	839	274,60	238,54
TN	3950	1142	1247	1903	1846	5579	363	269	1325	7863	2548,70	2354,65
FP	1428	531	478	506	925	3923	40	61	163	2739	1079,40	1222,06
FN	236	18	195	162	161	115	24	42	68	444	146,50	122,03
AUC	0,73	0,76	0,71	0,73	0,68	0,77	0,77	0,74	0,72	0,75	0,74	0,03
F,Score	0,33	0,19	0,45	0,42	0,33	0,19	0,54	0,55	0,37	0,35	0,37	0,12
Youden	0,37	0,47	0,31	0,39	0,29	0,39	0,51	0,42	0,39	0,40	0,39	0,06
TPR	0,64	0,79	0,59	0,60	0,62	0,81	0,61	0,60	0,50	0,65	0,64	0,09
FPR	0,27	0,32	0,28	0,21	0,33	0,41	0,10	0,18	0,11	0,26	0,25	0,09
J48												
TP	230	43	190	174	155	244	24	42	58	531	169,10	143,67
TN	4173	1005	1343	1741	2006	7176	314	258	1153	8357	2752,60	2726,60
FP	1205	668	382	668	765	2326	89	72	335	2245	875,50	773,71
FN	423	41	281	227	269	349	37	64	77	752	252,00	211,45
AUC	0,56	0,56	0,59	0,58	0,54	0,58	0,59	0,59	0,60	0,60	0,58	0,02
F,Score	0,22	0,11	0,36	0,28	0,23	0,15	0,28	0,38	0,22	0,26	0,25	0,08
Youden	0,13	0,11	0,18	0,16	0,09	0,17	0,17	0,18	0,20	0,20	0,16	0,04
TPR	0,35	0,51	0,40	0,43	0,37	0,41	0,39	0,40	0,43	0,41	0,41	0,04
FPR	0,22	0,40	0,22	0,28	0,28	0,24	0,22	0,22	0,23	0,21	0,25	0,05
Multilayer Perceptron												
TP	486	61	326	238	299	384	48	82	81	808	281,30	227,34
TN	3320	974	1272	1995	1638	6566	279	195	1200	7890	2532,90	2510,31
FP	2058	699	453	414	1133	2936	124	135	288	2712	1095,20	1023,78
FN	167	23	145	163	125	209	13	24	54	475	139,80	130,06
AUC	0,73	0,69	0,76	0,78	0,68	0,71	0,78	0,72	0,75	0,75	0,73	0,03
F,Score	0,30	0,14	0,52	0,45	0,32	0,20	0,41	0,51	0,32	0,34	0,35	0,12
Youden	0,36	0,31	0,43	0,42	0,30	0,34	0,48	0,36	0,41	0,37	0,38	0,05
TPR	0,74	0,73	0,69	0,59	0,71	0,65	0,79	0,77	0,60	0,63	0,69	0,07
FPR	0,38	0,42	0,26	0,17	0,41	0,31	0,31	0,41	0,19	0,26	0,31	0,09
Naive Bayes												
TP	502	64	250	273	329	379	43	68	87	836	283,10	235,90
TN	3042	1086	1348	1729	1441	6780	345	259	1147	7746	2492,30	2503,42
FP	2336	587	377	680	1330	2722	58	71	341	2856	1135,80	1047,15
FN	151	20	221	128	95	214	18	38	48	447	138,00	125,18
AUC	0,73	0,72	0,70	0,73	0,68	0,73	0,82	0,75	0,74	0,76	0,74	0,03
F,Score	0,29	0,17	0,46	0,40	0,32	0,21	0,53	0,56	0,31	0,34	0,36	0,12
Youden	0,33	0,41	0,31	0,40	0,30	0,35	0,56	0,43	0,42	0,38	0,39	0,07
TPR	0,77	0,76	0,53	0,68	0,78	0,64	0,70	0,64	0,64	0,65	0,68	0,07
FPR	0,43	0,35	0,22	0,28	0,48	0,29	0,14	0,22	0,23	0,27	0,29	0,10

detalhamento dos resultados dos algoritmos, considerando as métricas *TPR* e *FPR*, pode ser visto na Figura 11.

Considerando esta segunda análise, pode-se dizer que a presença de um projeto de grande porte entre os demais não trouxe ganhos significativos, pelo contrário, os resultados apontaram que o *cross-project-validation* utilizando somente projetos de tamanhos equiparados necessitam um menor esforço em eventuais testes.

A última análise foi realizada tomando somente um projeto como modelo e verificando qual a capacidade dele na predição dos outros projetos. Por possuir a maior quantidade de funções, o projeto torvalds/linux foi escolhido como modelo. A Tabela 11 é responsável por exibir os resultados obtidos nessa execução.

Um ponto a ser destacado nessa execução é quanto à quantidade de resultados onde o *TPR* e o *FPR* alcançaram valores muito parecidos. Isso acontece, por exemplo, quando ocorre uma quantidade muito baixa de FNs e muito alta de FPs, indicando que muitas

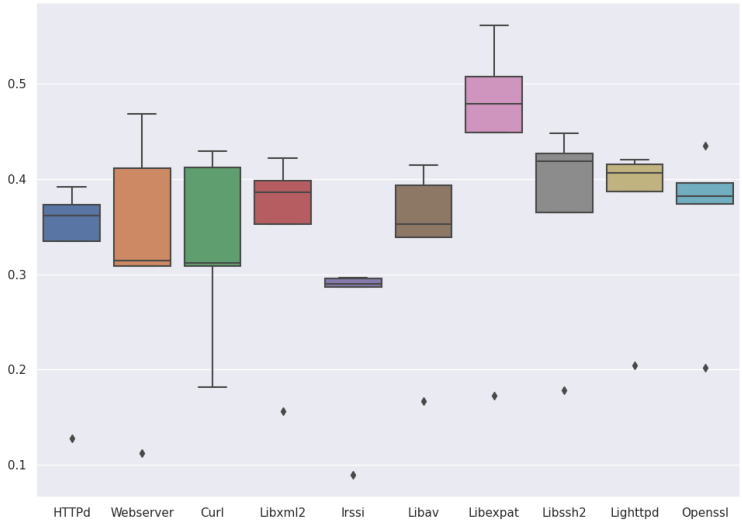

Figura 10 – Cross-project-validation parcial - Métrica *Youden-Index*

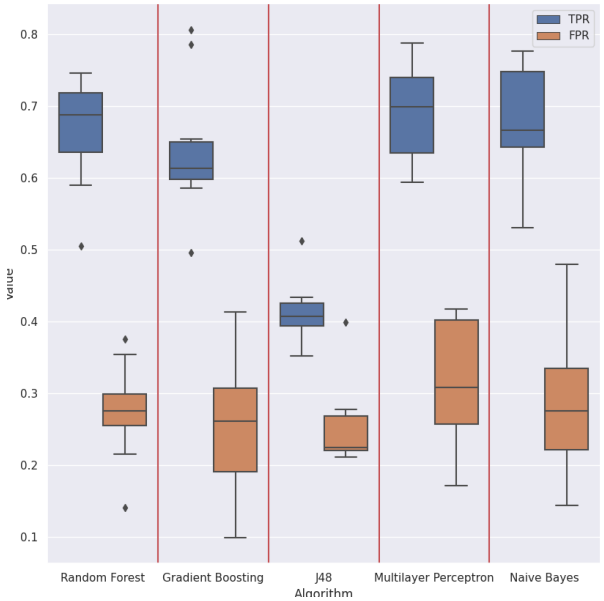

Figura 11 – Cross-project-validation parcial - Métricas *TPR* e *FPR*

funções vulneráveis foram apontadas, porém, um esforço adicional muito grande se fez necessário, já que diversas funções não vulneráveis foram classificadas erroneamente. Isso acontece também quando ocorre o inverso, obtendo *TPR* e *FPR* muito baixos, indicando que poucas funções foram apontadas como vulneráveis e assim, muitas delas deixaram de ser identificadas.

A análise da métrica *Youden* para essa execução reafirma a baixa qualidade da

Tabela 11 – *Modelo Kernel Linux* - Resultado

Métricas	HTTPd	Webserver	Curl	Libxml2	Irssi	Libav	Libexpat	Libssh2	Lighttpd	Openssl	Média	DP
P	653	84	471	401	424	593	61	106	135	1283	421,10	354,06
N	5378	1673	1725	2409	2771	9502	403	330	1488	10602	3628,10	3488,90
Random Forest												
TP	495	80	409	306	228	395	49	69	62	862	295,50	245,19
TN	2218	460	501	1023	1454	5444	122	225	1222	5823	1849,20	1986,22
FP	3160	1213	1224	1386	1317	4058	281	105	266	4779	1778,90	1563,10
FN	158	4	62	95	196	198	12	37	73	421	125,60	119,08
AUC	0,59	0,61	0,56	0,52	0,53	0,64	0,52	0,69	0,65	0,63	0,59	0,06
F,Score	0,23	0,12	0,39	0,29	0,23	0,16	0,25	0,49	0,27	0,25	0,27	0,10
Youden	0,17	0,23	0,16	0,19	0,06	0,24	0,11	0,33	0,28	0,22	0,20	0,08
TPR	0,76	0,95	0,87	0,76	0,54	0,67	0,80	0,65	0,46	0,67	0,71	0,14
FPR	0,59	0,73	0,71	0,58	0,48	0,43	0,70	0,32	0,18	0,45	0,51	0,17
Gradient Boosting												
TP	393	68	363	308	247	407	38	79	80	929	291,20	253,28
TN	4207	969	1020	1710	1915	6860	346	225	1214	7260	2572,60	2479,88
FP	1171	704	705	699	856	2642	57	105	274	3342	1055,50	1033,58
FN	260	16	108	93	177	186	23	27	55	354	129,90	107,22
AUC	0,73	0,74	0,74	0,77	0,66	0,76	0,78	0,75	0,73	0,76	0,74	0,03
F,Score	0,35	0,16	0,47	0,44	0,32	0,22	0,49	0,54	0,33	0,33	0,37	0,11
Youden	0,38	0,39	0,36	0,48	0,27	0,41	0,48	0,43	0,41	0,41	0,40	0,06
TPR	0,60	0,81	0,77	0,77	0,58	0,69	0,62	0,75	0,59	0,72	0,69	0,08
FPR	0,22	0,42	0,41	0,29	0,31	0,28	0,14	0,32	0,18	0,32	0,29	0,08
J48												
TP	532	60	309	221	262	492	43	77	80	998	307,40	284,06
TN	1307	589	589	1074	1398	2311	114	121	774	3176	1145,30	919,09
FP	4071	1084	1136	1335	1373	7191	289	209	714	7426	2482,80	2618,32
FN	121	24	162	180	162	101	18	29	55	285	113,70	81,51
AUC	0,52	0,53	0,49	0,49	0,56	0,53	0,49	0,54	0,55	0,53	0,52	0,02
F,Score	0,20	0,10	0,32	0,23	0,25	0,12	0,22	0,39	0,17	0,21	0,22	0,08
Youden	0,06	0,07	-0,00	-0,00	0,12	0,07	-0,01	0,09	0,11	0,08	0,06	0,05
TPR	0,81	0,71	0,66	0,55	0,62	0,83	0,70	0,73	0,59	0,78	0,70	0,09
FPR	0,76	0,65	0,66	0,55	0,50	0,76	0,72	0,63	0,48	0,70	0,64	0,10
Multilayer Perceptron												
TP	423	66	348	263	246	423	48	74	63	878	283,20	243,64
TN	4018	1086	1250	1786	1933	6636	284	241	1346	7953	2653,30	2542,63
FP	1360	587	475	623	838	2866	119	89	142	2649	974,80	963,39
FN	230	18	123	138	178	170	13	32	72	405	137,90	113,23
AUC	0,76	0,76	0,78	0,76	0,65	0,77	0,80	0,77	0,74	0,77	0,76	0,04
F,Score	0,35	0,18	0,54	0,41	0,33	0,22	0,42	0,55	0,37	0,37	0,37	0,11
Youden	0,39	0,43	0,46	0,40	0,28	0,41	0,49	0,43	0,37	0,43	0,41	0,06
TPR	0,65	0,79	0,74	0,66	0,58	0,71	0,79	0,70	0,47	0,68	0,68	0,09
FPR	0,25	0,35	0,28	0,26	0,30	0,30	0,30	0,27	0,10	0,25	0,27	0,06
Naive Bayes												
TP	512	64	420	252	293	424	49	69	68	1055	320,60	294,74
TN	2964	1078	726	1834	1661	6132	305	245	1248	5717	2191,00	2014,65
FP	2414	595	999	575	1110	3370	98	85	240	4885	1437,10	1529,19
FN	141	20	51	149	131	169	12	37	67	228	100,50	69,13
AUC	0,73	0,71	0,71	0,73	0,67	0,73	0,80	0,74	0,71	0,75	0,73	0,03
F,Score	0,29	0,17	0,44	0,41	0,32	0,19	0,47	0,53	0,31	0,29	0,34	0,11
Youden	0,34	0,41	0,31	0,39	0,29	0,36	0,56	0,39	0,34	0,36	0,38	0,07
TPR	0,78	0,76	0,89	0,63	0,69	0,72	0,80	0,65	0,50	0,82	0,73	0,11
FPR	0,45	0,36	0,58	0,24	0,40	0,35	0,24	0,26	0,16	0,46	0,35	0,12

transferência de conhecimento entre os projetos, alcançando resultados consideravelmente mais baixos, quando comparados aos obtidos nas análises anteriores. O modelo utilizado nessa execução se comportou melhor na predição do projeto libexpat/libexpat, tendo seu valor mais significante obtido através do algoritmo *Naive Bayes*. Já o projeto irssi/irssi foi responsável pelo pior resultado, embora não tenha obtido o menor mínimo entre todos os projetos analisados. A Figura 12 ilustra os resultados da métrica *Youden* obtidos nessa execução.

Diferente das execuções anteriores, agora os algoritmos apresentaram resultados mais parecidos para as métricas *TPR* e *FPR* (Figura 13). Percebe-se que os algoritmos *Random Forest* e *J48* são responsáveis pela maior similaridade dessas duas métricas. Já os algoritmos *Gradient Boosting*, *Multilayer Perceptron* e *Naive Bayes* obtiveram resultados melhores, sendo este segundo responsável por um menor esforço e o último, por uma maior efetividade, embora apresente uma maior dispersão de valores.

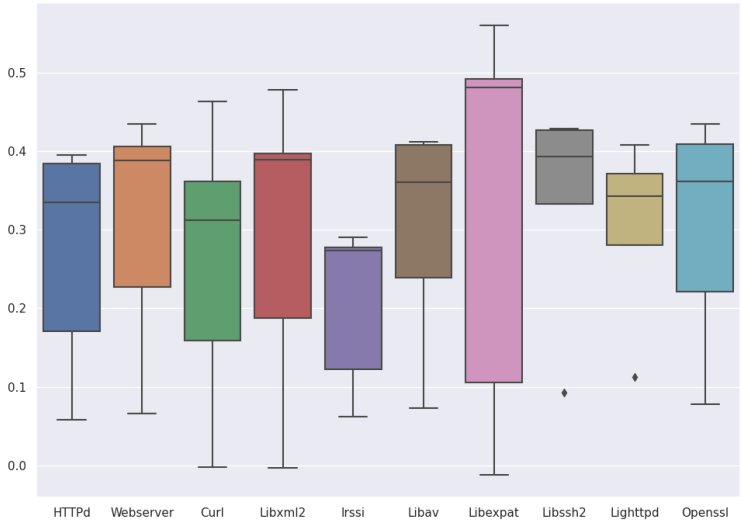

Figura 12 – Modelo Kernel Linux - Métrica *Youden-Index*

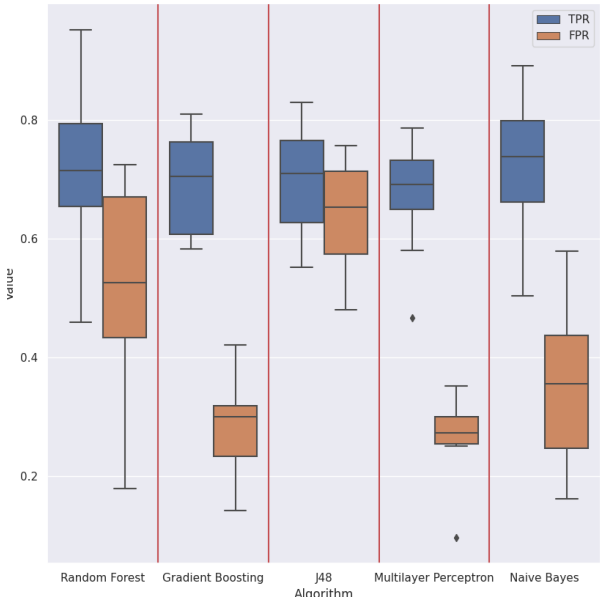

Figura 13 – Modelo Kernel Linux - Métricas *TPR* e *FPR*

Com base nos resultados obtidos nessa última execução, notou-se uma piora em termos de esforço e na qualidade do modelo ao se utilizar somente um projeto de grande porte na predição de projetos menores. Essa piora fica ainda mais evidente ao se observar os resultados obtidos pelos algoritmos *Random Forest* e *J48*.

4.3 RQ3: O modelo de predição tem um desempenho melhor do que as heurísticas de amostragem estado da arte?

Essa questão de pesquisa foi respondida comparando os resultados obtidos por cada uma das estratégias propostas anteriormente e o resultado original, obtido sem a interferência dessas estratégias. Ambos resultados foram obtidos utilizando a heurística de amostragem LSA, detalhada na Seção 2.5.

Tabela 12 – Cross-validation - Resultado amostragem

Projetos	Total	LSA			Random Forest			Gradient Boosting			J48			Multilayer Perceptron			Naive Bayes		
		Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão
httpd	653	420	582	1,38	304	346	1,13	318	377	1,18	242	187	0,77	320	342	1,06	306	453	1,48
webserver	84	26	84	3,23	6	65	10,83	1	52	52,00	1	24	24,00	6	62	10,33	8	58	7,25
curl	471	708	364	0,51	194	222	1,14	220	230	1,04	212	163	0,76	218	219	1,00	194	237	1,22
libxml2	401	438	240	0,54	256	206	0,80	136	160	1,17	122	107	0,87	136	147	1,08	106	145	1,36
irssi	424	52	406	7,81	46	275	5,97	32	177	5,53	26	91	3,50	44	219	4,97	40	266	6,65
libav	593	464	543	1,17	282	367	1,30	286	372	1,30	152	132	0,86	306	397	1,29	228	339	1,48
libexpat	61	20	56	2,80	14	50	3,57	12	40	3,33	10	21	2,10	12	46	3,83	8	39	4,87
libssh2	106	82	91	1,10	16	56	3,50	24	67	2,79	42	48	1,14	16	52	3,25	22	51	2,31
lighttpd1.4	135	96	113	1,17	62	116	1,87	60	101	1,68	24	45	1,87	34	71	2,08	44	80	1,82
openssl	1283	520	1188	2,28	420	891	2,12	376	778	2,06	320	488	1,52	394	672	1,70	420	944	2,24
linux	60292	7462	55375	7,42	4970	35523	7,14	4782	33709	7,04	3556	18291	5,14	4776	36878	7,72	5232	34030	6,50
Média	5863,91	935,27	5367,45	2,68	597,27	3465,18	3,58	567,91	3278,45	7,20	427,91	1781,55	3,87	569,27	3555,00	3,49	600,73	3331,09	3,38
DP	17214,98	2076,59	15816,90	2,47	1389,40	10140,11	3,03	1338,80	9625,16	14,29	994,44	5222,24	6,50	1337,10	10539,22	2,94	1470,29	9711,08	2,30

Assim como nas outras análises, os resultados foram comparados por meio da eficiência e esforço necessário. A eficiência se resume ao número de funções vulneráveis testadas em relação ao total de funções vulneráveis, já o esforço está relacionado ao número de variantes necessárias para realizar tal cobertura de testes. Como exemplo, na Tabela 12, considerando os resultados do LSA, devem ser testadas 582 das 653 funções vulneráveis existentes no projeto apache/httpd, porém um certo esforço se faz necessário, utilizando 420 variantes. Além disso, cada projeto foi comparado em termos da razão entre o número de funções testadas e variantes, indicando que quanto maior o valor obtido, mais funções uma única variante é capaz de testar.

O projeto cherokee/webserver foi o único que teve todas suas funções testadas utilizando o LSA, sendo recomendadas 26 variantes. Vale ressaltar que o resultado do LSA identifica todas as funções que não possuem expressões condicionais, diferentemente dos resultados gerados pelas estratégias de *Machine Learning* propostas, onde essas funções só são contabilizadas caso tenham sido apontadas como vulneráveis pelos modelos.

A primeira comparação foi realizada com os modelos treinados utilizando a estratégia de *cross-validation* (Tabela 12). Uma consideração importante é em relação à eficiência, onde se reafirma o destaque das *features* de Histórico e Desenvolvedor em relação às outras, assim como observado na seção 4.1. Para fins comparativos, foram utilizados os resultados utilizando todas as *features* (Complexidade + Vulnerabilidade + Histórico + Desenvolvedor).

Analisando os resultados, nota-se que 9 dos 11 projetos se comportaram melhor, considerando a razão, para a estratégia de *cross-validation*. Um destaque foi o projeto cherokee/webserver, onde foram recomendadas pelo LSA, 26 variantes para testar todas

as 84 funções vulneráveis, em contrapartida, apenas 1 variante para testar 52 funções vulneráveis, pelo resultado do *cross-validation*, utilizando o algoritmo *Gradient Boosting*.

Vale ressaltar que nenhum dos modelos conseguiu testar a mesma quantidade de funções que o LSA, isto está diretamente relacionado à qualidade desses modelos e das *features* presentes neles. O maior ganho ao se utilizar a estratégia de *Machine Learning* é em relação à redução do esforço, sendo necessário um número menor de variantes para conseguir uma boa cobertura de detecção. Isso fica mais evidente ao se analisar a média do número de variantes e de funções testadas para o LSA e, por exemplo, o *Random Forest*, onde este primeiro necessita de 935,27 variantes para testar 5.367,45 funções, obtendo uma razão de 2,68, enquanto o segundo necessita de 597,27 variantes para testar 3.465,18 funções, com uma razão de 3,58.

Tabela 13 – Cross-project-validation completo - Resultado amostragem

Projetos	Total	LSA			Random Forest			Gradient Boosting			J48			Multilayer Perceptron			Naive Bayes		
		Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão
httpd	653	420	582	1,38	298	384	1,28	306	371	1,21	298	387	1,29	302	363	1,20	282	364	1,29
webserver	84	26	84	3,23	1	59	59,00	10	62	6,20	8	50	6,25	8	61	7,62	12	64	5,33
curl	471	708	364	0,51	256	281	1,09	250	288	1,15	224	222	0,99	252	292	1,15	228	233	1,02
libxml2	401	438	240	0,54	142	159	1,11	130	156	1,20	230	107	0,46	132	155	1,17	232	157	0,67
irssi	424	52	406	7,81	38	209	5,50	46	280	6,08	34	205	6,02	34	206	6,05	40	266	6,65
libav	593	464	543	1,17	248	390	1,57	226	343	1,51	284	336	1,18	254	329	1,29	294	399	1,35
libexpat	61	20	56	2,80	14	43	3,07	14	42	3,00	10	33	3,30	14	45	3,21	10	44	4,40
libssh2	106	82	91	1,10	22	76	3,45	18	62	3,44	20	43	2,15	30	67	2,23	26	56	2,15
lighttpd1.4	135	96	113	1,17	42	73	1,73	50	83	1,66	38	70	1,84	50	64	1,28	58	88	1,51
openssl	1283	520	1188	2,28	328	836	2,54	350	770	2,20	338	716	2,11	346	763	2,20	422	983	2,32
linux	60292	7462	55375	7,42	4698	35935	7,64	4924	37494	7,61	3568	20569	5,76	4368	31688	7,25	5204	35136	6,75
Média	5863,91	935,27	5367,45	2,68	553,36	3495,00	8,00	574,91	3631,91	3,21	459,27	2067,09	2,85	526,36	3093,91	3,15	618,91	3435,45	3,04
DP	17214,98	2076,59	15816,90	2,47	1316,00	10260,79	16,24	1380,44	10710,03	2,24	990,88	5854,06	2,06	1220,77	9044,45	2,44	1456,14	10027,86	2,20

A segunda análise foi realizada entre o LSA e os resutados obtidos pela estratégia *Cross-project-validation* completo (Tabela 13). Com base nos resultados obtidos na análise da Seção 4.2, constatou-se que os algoritmos *Random Forest*, *Gradient Boosting* e *Naive Bayes* foram responsáveis pela maior média da métrica *TPR*. Além disso, *Multilayer Perceptron* obteve os melhores resultados para as métricas *FPR* e *Youden*.

Dos 11 projetos analisados, 9 das maiores razões foram obtidas pelo *Random Forest* (cherokee/webserver, libav/libav, libssh2/libssh2, openssl/openssl e torvalds/linux), 2 pelo *Gradient Boosting* (curl/curl e GNOME/libxml2), sendo o resultado deste primeiro projeto obtido também pelo *Multilayer Perceptron*, 1 pelo *J48* (lighttpd/lighttpd1.4), 1 pelo *Naive Bayes* (libexpat/libexpat) e 2 pelo LSA (apache/httpd e irssi/irssi). Assumindo que o *Random Forest* foi o projeto que obteve o maior número de melhores resultados, vale ressaltar que mesmo que os outros algoritmos não fossem considerados, ainda sim, a razão encontrada pelo *Random Forest* seria maior que o LSA.

Tabela 14 – Cross-project-validation parcial - Resultado amostragem

Projetos	Total	LSA			Random Forest			Gradient Boosting			J48			Multilayer Perceptron			Naive Bayes		
		Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão
httpd	653	420	582	1,38	324	428	1,32	288	361	1,25	194	204	1,05	332	428	1,28	300	442	1,47
webserver	84	26	84	3,23	1	52	52,00	6	66	11,00	12	43	3,58	1	61	61,00	16	64	4,00
curl	471	708	364	0,51	264	265	1,00	240	244	1,01	158	163	1,03	262	261	0,99	202	232	1,14
libxml2	401	438	240	0,54	138	162	1,17	138	140	1,01	132	101	0,76	132	172	1,30	246	163	0,66
irssi	424	52	406	7,81	32	197	6,15	36	246	6,83	40	138	3,45	46	281	6,10	42	312	7,42
libav	593	464	543	1,17	258	362	1,40	296	434	1,46	258	221	0,85	322	344	1,06	246	340	1,38
libexpat	61	20	56	2,80	14	33	2,35	14	33	2,35	10	23	2,30	14	45	3,21	10	39	3,90
libssh2	106	82	91	1,10	20	69	3,45	20	56	2,80	20	36	1,80	40	71	1,77	26	57	2,19
lighttpd1.4	135	96	113	1,17	52	89	1,71	42	64	1,52	28	56	2,00	48	77	1,60	44	84	1,90
openssl	1283	520	1188	2,28	366	827	2,25	388	792	2,04	308	528	1,71	362	756	2,08	384	791	2,05
Média	421,10	282,60	366,70	2,20	146,90	248,40	7,28	146,80	243,60	3,13	116,00	151,30	1,86	155,90	249,60	8,04	151,60	252,40	2,62
DP	354,06	240,06	329,61	2,06	135,19	230,84	14,98	136,36	224,69	3,09	104,84	142,46	0,96	139,35	210,58	17,71	131,98	222,09	1,91

Assim como na Seção 4.2, o projeto torvalds/linux foi removido com o objetivo de verificar o comportamento dos modelos. A Tabela 14 é responsável pela exibição dos resultados obtidos. No geral, os resultados não apresentaram ganhos ou perdas significantes quando comparados àqueles obtidos anteriormente.

Os projetos que se destacaram também se mantiveram parecidos aos obtidos utilizando o *Cross-project-validation* completo, sendo o resultado do projeto openssl/openssl a única divergência, onde a maior razão foi obtida pelo LSA. *Random Forest* foi o responsável por 3 dos 10 melhores resultados (cherokee/webserver, GNOME/libxml2 e libssh2/libssh2), *Gradient Boosting* por 1 (libav/libav), *J48* por 1 (lighttpd/lighttpd1.4), *Naive Bayes* por 3 (apache/httpd, curl/curl e libexpat/libexpat) e o LSA, por 2 (irssi/irssi e openssl/openssl).

Tabela 15 – Modelo Kernel Linux - Resultado amostragem

Projetos	Total	LSA			Random Forest			Gradient Boosting			J48			Multilayer Perceptron			Naive Bayes		
		Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão	Variantes	Funções	Razão
httpd	653	420	582	1,38	278	445	1,60	280	337	1,20	358	482	1,34	310	366	1,18	326	451	1,38
webserver	84	26	84	3,23	14	80	5,71	8	68	8,50	4	60	15,00	1	66	66,00	16	64	4,00
curl	471	708	364	0,51	310	341	1,10	272	301	1,10	296	255	0,86	288	276	0,95	306	339	1,10
libxml2	401	438	240	0,54	252	172	0,68	144	176	1,22	238	115	0,48	134	143	1,06	220	152	0,69
irssi	424	52	406	7,81	38	219	5,76	40	230	5,75	46	249	5,41	42	230	5,47	40	276	6,90
libav	593	464	543	1,17	214	371	1,73	270	369	1,36	332	457	1,37	310	386	1,24	284	383	1,34
libexpat	61	20	56	2,80	14	45	3,21	14	35	2,50	12	39	3,25	14	45	3,21	12	46	3,83
libssh2	106	82	91	1,10	8	62	7,75	22	67	3,04	56	64	1,14	40	63	1,57	26	59	2,26
lighttpd1.4	135	96	113	1,17	36	58	1,61	46	77	1,67	52	77	1,48	32	60	1,87	32	64	2,00
openssl	1283	520	1188	2,28	306	841	2,74	364	878	2,41	422	977	2,31	362	827	2,28	428	998	2,33
Média	421,10	282,60	366,70	2,20	147,00	263,40	3,19	146,00	253,80	2,88	181,60	277,50	3,27	153,30	246,20	8,49	169,00	283,20	2,59
DP	354,06	240,06	329,61	2,06	127,84	236,35	2,28	130,31	238,16	2,29	154,62	279,47	4,14	139,18	228,96	19,21	151,74	277,86	1,77

A última análise foi realizada utilizando como modelo somente o projeto torvalds/-linux. Uma observação relatada na Seção 4.2 é que os resultados obtidos por esse modelo apresentaram valores não muito bons para a maioria dos projetos, dificultando o balanceamento da eficiência e o esforço de maneira satisfatória. Isso fica ainda mais evidente ao se observar a Tabela 15, mais precisamente o resultado do projeto curl/curl. A predição deste projeto utilizando o modelo criado pelo torvalds/linux apresentou *TPR* e *FPR* altos, indicando uma alta eficiência, porém necessitando um grande esforço. Isso fica evidente na análise atual, onde são necessárias 310 variantes para testar 341 funções, utilizando o algoritmo *Random Forest*.

Considerando os 10 projetos analisados, o *Random Forest* obteve o melhor resultado para 5 deles (apache/httpd, curl/curl, libav/libav, libssh2/libssh2 e openssl/openssl), sendo que os algoritmos *Gradient Boosting* e *Naive Bayes* alcançaram o mesmo resultado para o projeto curl/curl. Além deste projeto, o *Gradient Boosting* obteve também o melhor resultado para o projeto GNOME/libxml2 e o *Naive Bayes*, para os projetos libexpat/-libexpat e lighttpd/lighttpd1.4. O algoritmo *Multilayer Perceptron* foi responsável pelo melhor resultado do projeto cherokee/webserver e o LSA, de apenas 1 projeto (irssi/irssi).

Considerando os resultados obtidos nas análises realizadas, é possível afirmar que o modelo de predição utilizando *Machine Learning* foi capaz de reduzir o esforço em eventuais testes, indicando um conjunto de amostras relativamente menor que os algoritmos de amostragem da heurística LSA e ainda assim, mantendo uma boa cobertura de detecção.

5 Trabalhos Relacionados

Ao longo dos anos, o interesse por produzir *software* mais seguros e livre de *bugs* tem aumentado significativamente. Isto se deve ao fato de que a maior parte dos dados importantes hoje são armazenados de forma digital, ficando expostos a constantes ameaças.

Diferentes estratégias foram propostas buscando aumentar a segurança e integridade de sistemas. A análise preditiva por meio do uso de abordagens de *Machine Learning* surgiu como uma das formas de facilitar o trabalho de engenheiros de *software* durante a inspeção do código e tem apresentado grandes resultados na área. Nessa seção, serão relacionados trabalhos com propostas relacionadas ao estudo em questão.

5.1 *Machine Learning* na predição de *bugs*

A existência de *bugs* de *software* afeta drasticamente a confiabilidade, qualidade e custo de manutenção do *software*. A predição de *bugs* é uma atividade essencial no desenvolvimento de *software*, visto que isto pode aumentar a satisfação do usuário e melhorar o desempenho geral do *software* (Hammouri, 2018).

Chappelly et al. (Chappelly, 2017) realizaram um estudo buscando determinar se as técnicas de *Machine Learning* que podem ser usadas na detecção de defeitos potenciais. Esse estudo foi realizado tomando as seguintes bases de dados: *Begbunch* e *OpenSolaris*. As *features* extraídas foram divididas em *n-grams*, representando o número que determinada operação ocorre em cada função, complexidade e *features* de texto, representando ocorrências de determinadas combinações de texto no código (por exemplo, '!='). Tomando a distância euclidiana como critério, foi realizada uma redução dos conjuntos de *features*, deixando aquelas melhor classificadas nesse contexto. Foi realizada então, uma análise comparativa entre os resultados obtidos pelo modelo criado e 3 mecanismos de análise estática: *Parfait*, *Splint* e *Uno*. Tanto utilizando o *Random Forest*, quanto experimentos com redes neurais, as análises mostraram que a abordagem mostra algum potencial para refinar a saída de outros mecanismos de análise estática, porém a eficácia cai muito quando esses dados de análise estática não estão disponíveis como recursos.

Hammouri et al. (Hammouri, 2018) também conduziram um estudo com o objetivo de analisar o desempenho e a capacidade dos algoritmos de *Machine Learning* na predição de *bugs* de *software*. Essa análise foi realizada em 3 bases de dados reais simples de *debug*, cada uma delas contendo o número de *bugs* ocorridos, número de processos de testes e o dia ocorrido. Foi realizada a estratégia de *cross validation*, utilizando o algoritmo

Naive Bayes, árvore de decisão e redes neurais artificiais. Os resultados obtiveram médias de *Recall*, *Precision* e *Accuracy* sempre superiores a 90%, revelando que as técnicas de *Machine Learning* são abordagens eficientes para prever *bugs* de *software*.

A principal relação entre esses estudos e o nosso, é em relação à viabilidade de algoritmos de *Machine Learning* na predição de instruções defeituosas inseridas pelo desenvolvedor, comprovada no estudo de Hammouri. Além disso, Chappelly apresentou conjuntos de *features* que serviram de inspiração para a criação dos grupos utilizados pelos nossos modelos.

5.2 *Machine Learning* na predição de vulnerabilidades

Alguns trabalhos se propuseram a utilizar *Machine Learning* para encontrar vulnerabilidades em sistemas. É o caso do *framework LEOPARD*, proposto por Xiaoning Du et al. (Du, 2019), cujo objetivo é identificar funções potencialmente vulneráveis em aplicações C/C++. Para tal, *LEOPARD* utiliza dois conjuntos de *features*: complexidade e vulnerabilidade. Neste primeiro conjunto são colocadas as *features* relacionadas à complexidade ciclomática e estruturas de repetição. Já no segundo, são colocadas as *features* referentes à dependência da função, ponteiros e dependências entre estruturas de controle. Os resultados obtidos por Du, realizando experimentos com 11 projetos reais, indicaram uma cobertura de 74% das funções vulneráveis. Os grupos de *features* Complexidade e Vulnerabilidade utilizados em nossa estratégia foram baseados nesse estudo.

Outro estudo que se propôs a avaliar a influência de conjuntos de *features* foi realizado por Sara Moshrari e Ashkan Sami (Moshtari; Sami, 2016). Neste estudo, foram avaliados o poder de previsibilidade de *features* de complexidade, acoplamento e um novo conjunto de *features* de acoplamento, denominado *Included Vulnerable Header* (IVH). Como forma de avaliação, foi realizada uma estratégia de *cross-project validation* em 5 projetos de diferentes linguagens, tamanhos e contextos de uso: *Eclipse*, *Apache Tomcat*, *Mozilla Firefox*, *Linux Kernel* e *OpenSCADA*. A base de dados foi divida não por funções, como em nosso trabalho, mas por arquivos, utilizando 15 categorias de vulnerabilidades relatadas pelo *CWE* (CWE, 2021). Quanto às *features*, foram selecionadas 11 de complexidade, 4 de acoplamento e 6 *features* relacionas ao novo conjunto (*IVH*). Os resultados considerando a métrica *Recall* (Tabela 16) indicaram que as *features* de complexidade são preditores de vulnerabilidade mais fortes do que as *features* de acoplamento, além de apontar um ganho ao utilizar o conjunto *IVH* juntamente às outras *features*.

Tabela 16 – *Recall* - Estudo *Sara Moshrari e Ashkan Sami*

Conjunto de *features*	NaiveBayes	ClassificationviaClustering	ThresholdSelector	RandomTree
Acoplamento	16,1	39,4	20,4	9,6
Acoplamento + IVH	48,5	47,7	42,8	2
Complexidade	25,1	60,9	26,6	7,9
Complexidade + IVH	58,9	87,4	44,8	30,5

Shin et al. (Shin, 2011) realizaram um estudo para investigar a aplicabilidade de três categorias de *features* de *software* para construir modelos de predição de vulnerabilidade: complexidade, rotatividade de código e atividade de desenvolvedor. Para compor o conjunto de *features*, foram selecionadas 14 de complexidade, 3 *features* de rotatividade e 11 de desenvolvedor. Foram realizados dois estudos de caso, o primeiro utilizando o *Mozilla Firefox*, cujas vulnerabilidades foram coletadas de *Mozilla Foundation Security Advisories* (Mozilla, 2021) e atribuídas aos arquivos do projeto. O segundo estudo de caso foi realizado utilizando o *kernel* do *Red Hat Enterprise Linux* e suas vulnerabilidades coletadas do *Bugzilla* e do sistema de gerenciamento de pacote do *Red Hat (RPM)*. A qualidade das *features* foram avaliadas em termos de probabilidade de detecção (PD) e probabilidade de alarme falso (PF). Os resultados mostraram que os modelos com *features* de complexidade por si só forneceram o desempenho de predição mais fraco. Por outro lado, as *features* de rotatividade de código, atividade do desenvolvedor e também os três conjuntos de *features* combinados podem reduzir potencialmente o esforço de inspeção de vulnerabilidade em comparação com uma seleção aleatória de arquivos. Esse trabalho foi de grande influência para a criação dos grupos de *features* Histórico e Desenvolvedor.

5.3 Estratégias de *Sampling*

Diversos estudos relataram que os problemas relacionados à configurabilidade são mais difíceis de detectar do que os problemas que aparecem em todas as variantes, visto que a variabilidade aumenta a complexidade. Medeiros et al. (Medeiros, 2015) realizaram um estudo em 15 projetos *open-source* configuráveis, escritos em C, buscando entender como os desenvolvedores introduzem problemas relacionados à variabilidade, o número de opções de configuração envolvidas e por quanto tempo esses problemas permanecem nos arquivos de origem.

A estratégia consiste em realizar um *parse* dos arquivos de cada projeto utilizando o *parser variability-aware TypeChef* (Kästner, 2021) e gerar uma *AST*. Baseando-se nela, é possível apontar as seguintes categorias de falhas: variáveis não declaradas, variáveis não utilizadas, funções não declaradas e funções não utilizadas. Analisando os 15 projetos, foram detectadas 16 falhas (2 variáveis e 15 funções não declaradas) e 23 avisos (16 variáveis e 7 funções não utilizadas), ambos relacionados à variabilidade. Os resultados revelaram que mais de 87% dos problemas de variabilidade envolvem duas ou menos variantes, além de que 73% desses problemas estão relacionados à adição de novos códigos, como arquivos ou funções.

Uma consideração importante é que problemas relacionados a erros de sintaxe são, em sua maioria, introduzidos em modificações de código. No geral, o algoritmo de *sampling pair-wise* (Perrouin, 2010) consegue detectar mais de 87% dos problemas encontrados no

estudo, porém existem problemas que envolvem mais de duas opções de configuração, se fazendo necessário o uso de algoritmos de amostragem mais complexos. Outra evidência encontrada nesse estudo foi que os problemas relacionados à variabilidade permanecem nos arquivos de origem por vários anos, enquanto problemas que aparecem em todas as variantes são normalmente corrigidos em alguns dias. O estudo realizado por Medeiros foi de suma importância para entendermos características intrínsecas de sistemas configuráveis, mais precisamente de problemas relacionados a esses sistemas.

Shu et. al. (Shu, 2020) realizaram um estudo buscando reduzir o tamanho das amostras de variantes utilizadas em predições de desempenho de sistemas configuráveis. Para isso, foi proposto o modelo *Perf-AL*, um *framework* que utiliza aprendizagem adversária onde por meio de da competição e adaptação iterativa do gerador e discriminador, é tido como resultado um modelo de predição melhor que outras abordagens de *Machine Learning*. Foi realizada uma comparação com 2 preditores de desempenho estado-da-arte: *DECART* e *DeepPerf*, utilizando 7 projetos de diferentes tamanhos e linguagens. Os resultados experimentais demonstraram que *Perf-AL* supera esses outros preditores, utilizando amostras menores. Embora não seja relacionado à predição de vulnerabilidades, esse estudo valida que é possível obter um conjunto de amostragem relativamente pequeno, mantendo um bom resultado.

6 Conclusão

Esse trabalho abordou conceitos de vulnerabilidades em *software*, mais especificamente em sistemas configuráveis, onde o número de variantes pode crescer exponencialmente, inviabilizando o uso de técnicas tradicionais para encontrar tais vulnerabilidades. A nossa proposta foi, utilizando estratégias de *Machine Learning*, reduzir o tamanho da amostra de variantes, enquanto se mantém uma boa cobertura de detecção.

Foram coletadas 53 *features* referentes à Complexidade, Vulnerabilidade, Histórico e Desenvolvedor, para 11 projetos escritos em C. Essas *features* foram selecionadas baseando-se em trabalhos presentes na literatura e utilizadas no treinamento dos modelos. Essas categorias de *features* foram adicionadas sequencialmente ao modelo buscando encontrar qual delas apresenta melhor comportamento na predição de vulnerabilidades. Os resultados mostraram que as *features* de Histórico e Desempenho apresentaram um ganho mais significativo ao serem inseridas no modelo, conseguindo aprimorar a eficiência, diminuindo o número de FNs e redução do esforço, diminuindo o número de FPs.

A estratégia foi submetida também a uma análise visando descobrir se os modelos de predição poderiam ser transferidos entre projetos de *software*, ou seja, treinar o modelo em um projeto e testá-lo em outro. Foram realizadas três execuções, alternando entre os tamanhos das bases de treinamento e teste. Os resultados foram satisfatórios ao se treinar o modelo utilizando N-1 projetos para testá-lo no projeto restante, porém esse mesmo comportamento não foi obtido ao se treinar com apenas um e testar os outros N-1 projetos.

Por fim, a última análise buscou comparar o número e consequentemente, a cobertura de testes alcançada pelas variantes indicadas pela nossa estratégia àquelas indicadas pela heurística LSA, onde são combinados três algoritmos de amostragem: *One-enabled*, *One-disabled* e *Most-enabled-disabled*. Os resultados foram comparados considerando o número de funções vulneráveis que uma única variante é capaz de testar. Por mais que o conjunto de amostras recomendado não consiga uma cobertura similar à obtida pelo LSA, nossos resultados demonstram ser possível reduzir o tamanho da amostra de variantes, enquanto se mantém uma boa cobertura de detecção.

Referências

Medeiros, F. An approach to safely evolve program families in c. In: *Proceedings of the Companion Publication of the 2014 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*. New York, NY, USA: Association for Computing Machinery, 2014. (SPLASH '14), p. 25–27. ISBN 9781450332088. Disponível em: <https://doi.org/10.1145/2660252.266025>. Citado 3 vezes nas páginas 9, 19 e 20.

Brabrand, C.; Ribeiro, M.; Tolêdo, T.; Winther, J.; Borba, P. Intraprocedural dataflow analysis for software product lines. Springer, p. 73–108, 2013. Citado na página 15.

Ferreira, G.; Malik, M.; Kastner, C.; Pfeffer, J.; Apel, S. Do ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel. 2016. Citado na página 15.

Sampaio, L.; Garcia, A. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software*, v. 113, p. 337–361, 2016. Citado na página 15.

Liebig, J.; Von Rhein, A.; Kästner, C.; Apel, S.; Dorre, J.; Lengauer, C. Large-scale variability-aware type checking and dataflow analysis. 2012. Citado na página 15.

Medeiros, F.; Kästner, C.; Ribeiro, M.; Gheyi, R.; Apel, S. A comparison of 10 sampling algorithms for configurable systems. p. 643–654, 2016. Citado 3 vezes nas páginas 15, 27 e 28.

GITHUB. *GitHub*. 2021. Disponível em: <https://github.com>. Citado 2 vezes nas páginas 16 e 29.

SciTools. *Understand*. 2021. Disponível em: <https://www.scitools.com>. Citado 2 vezes nas páginas 16 e 31.

Joern. *Joern*. 2021. Disponível em: <https://joern.io>. Citado 2 vezes nas páginas 16 e 31.

Du, X.; Chen, B.; Li, Y.; Guo, J.; Zhou, Y.; Liu, Y.; Jiang, Y. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2019. p. 60–71. Citado 3 vezes nas páginas 16, 37 e 67.

NVD. *National Vulnerability Database*. 2021. Disponível em: <https://nvd.nist.gov>. Citado 2 vezes nas páginas 16 e 39.

Easterbrook, S.; Singer, J.; Storey, M.-A.; Damian, D. Selecting empirical methods for software engineering research. *Empirical Software Engineering - ESE*, p. 285–311, 01 2008. Citado na página 17.

Woodside, A.; Wilson, E. Case study research for theory-building. *Journal of Business e Industrial Marketing*, v. 18, p. 493–508, 12 2003. Citado na página 17.

WOHLIN, C.; RUNESON, P.; HOST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLéN, A. *Experimentation in Software Engineering*. [S.l.: s.n.], 2012. Citado na página 17.

McIlroy, M. D. Mass-produced software components. In: . [S.l.]: Springer-Verlag, 1968. Citado na página 19.

Ali Babar, M.; Chen, L.; Shull, F. Managing variability in software product lines. In: . [S.l.: s.n.], 2010. v. 27, p. 89 – 91, 94. Citado 2 vezes nas páginas 19 e 20.

Reisner, E.; Song, C.; Ma, K.; Foster, J. S.; Porter, A. Using symbolic evaluation to understand behavior in configurable software systems. v. 1, p. 445–454, 2010. Citado na página 19.

Pohl, K.; Böckle, G.; van der Linden, F. *Software Product Line Engineering: Foundations, Principles, and Techniques*. [S.l.: s.n.], 2005. ISBN 3540243720. Citado na página 20.

Michaelis. *Dicionário Brasileiro da Língua Portuguesa*. 2021. Disponível em: <https://michaelis.uol.com.br>. Citado na página 21.

Anley, C.; Koziol, J.; Linder, F.; Richarte, G. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. USA: John Wiley amp; Sons, Inc., 2007. ISBN 047008023X. Citado na página 21.

Candal-Vicente, I.; Castro-Gonzáles, S.; García-Cortés, J. Evaluation of vulnerabilities in computer systems users. *Journal of Information System Security*, v. 13, p. 3–22, 08 2017. Citado na página 21.

Compuquip. *Computer Security Vulnerabilities*. 2021. Disponível em: <https://www.compuquip.com/blog/computer-security-vulnerabilities>. Citado na página 21.

OWASP. *OWASP Top 10*. 2020. Disponível em: <https://owasp.org/www-project-top-ten>. Citado na página 21.

CWE. *Common Weakness Enumeration*. 2021. Disponível em: <https://cwe.mitre.org>. Citado 3 vezes nas páginas 21, 39 e 67.

Mitre. *The Mitre Corporation*. 2021. Disponível em: <https://www.mitre.org>. Citado na página 21.

Li, J.; Zhao, B.; Zhang, C. Fuzzing: a survey. *Cybersecurity*, v. 1, 12 2018. Citado na página 22.

Müller, A.; Guido, S. *Introduction to Machine Learning with Python: A Guide for Data Scientists*. [S.l.: s.n.], 2016. Citado 6 vezes nas páginas 22, 23, 25, 26, 31 e 44.

Dey, A. Machine learning algorithms : A review. 2016. Citado na página 22.

BURKOV, A. *The Hundred-Page Machine Learning Book*. Andriy Burkov, 2019. ISBN 9781999579517. Disponível em: <https://books.google.com.br/books?id=0jbxwQEACAA>. Citado na página 23.

Alenezi, M.; Abunadi, I. Evaluating software metrics as predictors of software vulnerabilities. *International Journal of Security and Its Applications*, v. 9, p. 231–240, 10 2015. Citado na página 24.

Gonzalez-Recio, O.; Jimenez-Montero, J.; Alenda, R. The gradient boosting algorithm and random boosting for genome-assisted evaluation in large data sets. *Journal of Dairy Science*, v. 96, n. 1, p. 614–624, 2013. ISSN 0022-0302. Disponível em: <https://www.sciencedirect.com/science/article/pii/S002203021200778>. Citado na página 24.

Marsland, S. *Machine Learning: An Algorithmic Perspective, Second Edition*. 2nd. ed. [S.l.]: Chapman amp; Hall/CRC, 2014. ISBN 1466583282. Citado 2 vezes nas páginas 25 e 26.

Stefan, M.; Panduru, N.; Ion, D. Estimation of youden index and its associated optimal cut-point when the parameters of gamma biomarkers are estimated by the method of moments. In: . [S.l.: s.n.], 2011. v. 12. Citado na página 26.

Oh, J.; Gazzillo, P.; Batory, D. T-wise coverage by uniform sampling. In: . New York, NY, USA: Association for Computing Machinery, 2019. (SPLC '19), p. 84–87. ISBN 9781450371384. Disponível em: <https://doi.org/10.1145/3336294.334235>. Citado na página 28.

APACHE/HTTPD. *apache/httpd*. 2020. Disponível em: <https://github.com/apache-/http>. Citado na página 29.

CHEROKEE/WEBSERVER. *cherokee/webserver*. 2020. Disponível em: <https:/-/github.com/cherokee/webserve>. Citado na página 29.

CURL/CURL. *curl/curl*. 2020. Disponível em: <https://github.com/curl/cur>. Citado na página 29.

GNOME/LIBXML2. *GNOME/libxml2*. 2020. Disponível em: <https://github.com-/GNOME/libxml>. Citado na página 29.

IRSSI/IRSSI. *irssi/irssi*. 2020. Disponível em: <https://github.com/irssi/irss>. Citado na página 29.

LIBAV/LIBAV. *libav/libav*. 2020. Disponível em: <https://github.com/libav/liba>. Citado na página 29.

LIBEXPAT/LIBEXPAT. *libexpat/libexpat*. 2020. Disponível em: <https://github.com-/libexpat/libexpa>. Citado na página 30.

LIBSSH2/LIBSSH2. *libssh2/libssh2*. 2020. Disponível em: <https://github.com/libssh2-/libssh>. Citado na página 30.

LIGHTTPD/LIGHTTPD1.4. *lighttpd/lighttpd1.4*. 2020. Disponível em: <https://github-.com/lighttpd/lighttpd1.>. Citado na página 30.

OPENSSL/OPENSSL. *openssl/openssl*. 2020. Disponível em: <https://github.com-/openssl/openss>. Citado na página 30.

TORVALDS/LINUX. *torvalds/linux*. 2020. Disponível em: <https://github.com-/torvalds/linu>. Citado na página 30.

Younis, A.; Malaiya, Y.; Anderson, C.; Ray, I. To fear or not to fear that is the question: Code characteristics of a vulnerable functionwith an existing exploit. In: . New York, NY, USA: Association for Computing Machinery, 2016. (CODASPY '16). ISBN 9781450339353. Disponível em: <https://doi.org/10.1145/2857705.285775>. Citado na página 31.

Bosu, A. Characteristics of the vulnerable code changes identified through peer code review. In: *Companion Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE Companion 2014), p. 736–738. ISBN 9781450327688. Disponível em: <https://doi.org/10.1145/2591062.259120>. Citado 3 vezes nas páginas 32, 35 e 39.

Spadini, D. *Pydriller*. 2021. Disponível em: <https://github.com/ishepard/pydrille>. Citado na página 32.

Mundfrom, D.; Whitcomb, A. Imputing missing values: The effect on the accuracy of classification. In: . [S.l.: s.n.], 1998. Citado na página 41.

Brownlee, J. *Data Preparation for Machine Learning*. [S.l.: s.n.], 2020. Citado na página 41.

Song, Q.; Guo, Y.; Shepperd, M. A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineering*, v. 45, n. 12, p. 1253–1269, 2019. Citado na página 42.

Chawla, N. V.; Bowyer, K. W.; Hall, L. O.; Kegelmeyer, W. P. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, AI Access Foundation, El Segundo, CA, USA, v. 16, n. 1, p. 321–357, jun. 2002. ISSN 1076-9757. Citado na página 42.

Scikit-learn. *Scikit-learn*. 2021. Disponível em: <https://scikit-learn.org>. Citado na página 42.

Aniche, M.; Maziero, E.; Durelli, R.; Durelli, V. H. S. The effectiveness of supervised machine learning algorithms in predicting software refactoring. In: . [S.l.: s.n.], 2020. abs/2001.03338. Citado na página 42.

Lei, S. A feature selection method based on information gain and genetic algorithm. In: *2012 International Conference on Computer Science and Electronics Engineering*. [S.l.: s.n.], 2012. v. 2, p. 355–358. Citado na página 50.

Hammouri, A.; Hammad, M.; Alnabhan, M. M.; Alsarayrah, F. Software bug prediction using machine learning approach. In: . [S.l.: s.n.], 2018. v. 9. Citado na página 66.

Chappelly, T.; Cifuentes, C.; Krishnan, P.; Gevay, S. Machine learning for finding bugs: An initial report. In: *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. [S.l.: s.n.], 2017. p. 21–26. Citado na página 66.

Moshtari, S.; Sami, A. Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction. In: . [S.l.: s.n.], 2016. p. 1415–1421. Citado na página 67.

Shin, Y.; Meneely, A.; Williams, L.; Osborne, J. A. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. In: . [S.l.: s.n.], 2011. v. 37, n. 6, p. 772–787. Citado na página 68.

Mozilla. *Mozilla Foundation Security Advisories*. 2021. Disponível em: <http://www-.mozilla.org/security/announce>. Citado na página 68.

Medeiros, F.; Rodrigues, I.; Ribeiro, M.; Teixeira, L.; Gheyi, R. An empirical study on configuration-related issues: Investigating undeclared and unused identifiers. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. New York, NY, USA: Association for Computing Machinery, 2015. (GPCE 2015), p. 35–44. ISBN 9781450336871. Disponível em: <https://doi.org/10.1145/2814204.281420>. Citado na página 68.

Kästner, C. *TypeChef*. 2021. Disponível em: <https://github.com/ckaestne/TypeChe>. Citado na página 68.

Perrouin, G.; Sen, S.; Klein, J.; Baudry, B.; Le Traon, Y. Automated and scalable t-wise test case generation strategies for software product lines. In: . [S.l.: s.n.], 2010. Citado na página 68.

Shu, Y.; Sui, Y.; Zhang, H.; Xu, G. Perf-al: Performance prediction for configurable software through adversarial learning. In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. New York, NY, USA: Association for Computing Machinery, 2020. (ESEM '20). ISBN 9781450375801. Disponível em: <https://doi.org/10.1145/3382494.341067>. Citado na página 69.