UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS GRADUAÇÃO EM INFORMÁTICA

Henrique Ferreira Alves

UM ESTUDO DA RELAÇÃO ENTRE COMPLEXIDADE ESTRUTURAL E VULNERABILIDADES DE SOFTWARE

Maceió - AL
2017

HENRIQUE FERREIRA ALVES

UM ESTUDO DA RELAÇÃO ENTRE COMPLEXIDADE ESTRUTURAL E VULNERABILIDADES DE SOFTWARE

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas, como requisito para obtenção do grau de Mestre em Informática.

Orientador: Prof. Dr. Baldoino Fonseca
Coorientador: Prof. Dr. Nuno Antunes

Maceió - AL
2017

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Bibliotecária Responsável: Janaina Xisto de Barros Lima

A474e Alves, Henrique Ferreira.
Um estudo da relação entre complexidade estrutural e vulnerabilidades de software / Henrique Ferreira Alves. – 2017.
92 f. : il.

Orientador: Baldoino Fonseca dos Santos Neto.
Coorientador: Nuno Manoel dos Santos Antunes.
Dissertação (mestrado em Informática) – Universidade Federal de Alagoas. Instituto de Computação. Programa de Pós-Graduação em Informática. Maceió, 2017.

Bibliografia: f. 89-92.

1. Banco de dados. 2. Mineração de dados (computação). 3. Métrica de software. 4. Aprendizagem de máquina. 5. Software – Vulnerabilidade. I. Título.

CDU: 004.412

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Programa de Pós-Graduação em Informática – PpgI
Instituto de Computação
Campus A. C. Simões BR 104-Norte Km 14 BL 12 Tabuleiro do Martins
Maceió/AL - Brasil CEP: 57.072-970 | Telefone: (082) 3214-1401

Membros da Comissão Julgadora da Dissertação de Henrique Ferreira Alves, intitulada: "Um Estudo da Relação entre Complexidade Estrutural e Vulnerabilidade de Software", apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas em 12 de setembro de 2017, às 08h30min, na Sala Alan Turing, no Instituto de Matemática da UFAL.

COMISSÃO JULGADORA

Prof. Dr. Baldoino Fonseca dos Santos Neto
UFAL – Instituto de Computação
Orientador

Prof. Dr. Nuno Manuel dos Santos Antunes
Universidade de Coimbra
Coorientador

Prof. Dr. Patrick Henrique da Silva Brito
UFAL – Instituto de Computação
Examinador

Prof. Dra. Regina Lúcia de Oliveira Moraes
UNICAMP – Universidade Estadual de Campinas
Examinador

Aos amigos e familiares que contribuíram para realização desta pesquisa. Aos meus orientadores: Baldoino Fonseca e Nuno Antunes. E principalmente à Deus, pela saúde, força e oportunidade.

AGRADECIMENTOS

Uma dissertação não é fruto de um empenho solitário. Do início da pesquisa até suas conclusões finais, várias pessoas contribuíram para esclarecimentos, críticas e investigações para produção do conhecimento estabelecido neste trabalho. Quero, portanto, agradecer a todas as pessoas que direta e indiretamente me ajudaram na realização desta pesquisa de mestrado.

Começo assim a agradecer aos meus orientadores, Prof. Dr. Baldoino Fonseca e Prof. Dr. Nuno Antunes, pela missão, esforço e dedicação de me auxiliar na realização deste trabalho, que desde o começo, contribuíram e apostaram com entusiasmo na tarefa proposta pela pesquisa.

Agradeço também aos Professores: Dr. Marco Vieira e Dr. Alessandro Garcia pela oportunidade de expandir os ideais deste trabalho a comunidades externas que puderam acrescentar e evoluir tanto para o trabalho, como para crescimento pessoal. Agradeço o acompanhamento da banca examinadora na realização da qualificação com sugestões e incentivo dos Professores Dr. Alessandro Garcia e Dr. Patrick Brito.

Agradeço aos meus amigos e colegas que conviveram e ajudaram no caminho trilhado para realização deste mestrado, em especial Marcus Piancó, João Lucas, Gabriela Nunes e Luis Ventura. Por fim, agradeço à minha família que me apoiou de todas as formas para realização do mestrado.

RESUMO

A necessidade de desenvolver, gerir e evoluir os produtos de software está em alta, isso se deve à dependência que as pessoas têm das novas tecnologias. O uso de programas computacionais já está impregnado nas atividades cotidianas. Entretanto, há um problema que os desenvolvedores e gestores enfrentam e que compromete a integridade das funcionalidades dos sistemas, as vulnerabilidades de software. Pesquisas mostram que a maneira mais barata de se evitar uma vulnerabilidade é a utilização de inspeção de código. Entretanto, não é possível realizar esse processo em todo o software, existe uma grande quantidade de algoritmos e fluxos de tarefas em um software que inviabilizam tal atividade. Como estratégia, desenvolvedores focam seus olhos em partes de código mais relevantes, para isso contam com estratégias que identifiquem as partes mais complexas do software. A literatura estuda o uso de técnicas como o uso de métricas de software para direcionar a atenção dos revisores às partes que, supostamente, seriam as mais propensas a conter vulnerabilidade. Em nosso trabalho, vamos realizar um estudo exploratório em sete projetos *open source*: Mozilla Firefox, Xen Hipervisor, Linux Kernel, Httpd Server, Glibc, Tomcat e Derby, no qual vamos extrair métricas de software de suas classes, arquivos e funções a fim de relacioná-las às vulnerabilidades. A intenção principal é contribuir com novas pesquisas, disponibilizar dados e guiar novas investigações com informações relevantes dessa relação.

Palavras-chaves: Vulnerabilidades. Banco de dados. Mineração de dados. Aprendizagem de máquina.

ABSTRACT

The need to develop, manage and evolve the software is on the rise, this is due to dependence that people have on new technologies. The use of computer programs already are steeped in daily activities. However there is a problem that developers and managers face and that compromises the integrity of the features of theses programs, software vulnerabilities. Research shows that the cheapest way to avoid vulnerability is the use of code inspection, however it is not possible to carry out this process across the whole software, there is an exorbitant amount of algorithms and job streams in software that prevent such activity. As a strategy, developers focus their eyes on the most important pieces of code for that have strategies to identify the most complex pieces of software. The literature studies the use of techniques such as the use of software metrics to direct the attention of reviewers to the parties that are supposed to be the most likely to contain vulnerabilities. In our work, we will conduct an exploratory study in seven open source projects: Mozilla Firefox, Xen Hypervisor, Linux Kernel, Httpd Server, Glibc, Tomcat and Derby, which software metrics from their classes, files and functions were extracted in order to relate them to vulnerabilities. The main intention is to contribute with new research, to provide data and to guide new investigations with relevant information of this relation.

Key-words: Vulnerabilities. Dataset. Data Mining. Machine Learning. Evaluation.

LISTA DE PUBLICAÇÕES

ALVES, Henrique; FONSECA, Baldoino; ANTUNES, Nuno. Software Metrics and Security Vulnerabilities: Dataset and Exploratory Study. In: **12th European Dependable Computing Conference (EDCC 2016)**. Gothenburg, Sweden: [s.n.], 2016

ALVES, Henrique; FONSECA, Baldoino; ANTUNES, Nuno. Experimenting Machine Learning Techniques to Predict Vulnerabilities. In: **Dependable Computing (LADC), 2016 Seventh Latin-American Symposium on**. IEEE, 2016. p. 151-156.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Ataques a sistemas computacionais por *hackers* (ataques *hackers*) podem comprometer o uso de um sistema, violar privacidade de usuários e gerar perdas financeiras irrecuperáveis. Os danos financeiros causados às empresas por causa de ataques hackers cercam 400 bilhões de dólares (LOSSES, 2014). Estimativas indicam que se esse crescente aumento de ataques não cessar, as perdas financeiras podem atingir 3 trilhões de dólares em 2020 (KAPLAN; WEINBERG; CHINN, 2014). Um ataque acontece quando uma pessoa mal-intencionada se beneficia de uma vulnerabilidade do sistema e realiza atividades indesejáveis. Essas atividades podem comprometer a proteção de informações ou mesmo interromper serviços providos pelo sistema.

Vulnerabilidades são brechas no software que possibilitam acessos ou comportamentos inesperados do sistema. Um ataque só acontecerá se antes existir uma vulnerabilidade a ser explorada. Por isso, deve-se evitar ao máximo que produtos de software sejam implantados sem os devidos cuidados, garantindo que os requisitos mínimos de segurança foram atendidos. Isso porque é impossível assegurar, com 100% de certeza, que seu código está livre dessas falhas, de modo que pesquisas já tentam prever a quantidade de vulnerabilidades dependendo do sistema de software (ALHAZMI; MALAIYA; RAY, 2007). Porém, certas práticas de desenvolvimento podem propiciar maior grau de segurança nos produtos de software implantados. Por exemplo, o uso de certos padrões e protocolos de segurança podem reduzir vulnerabilidades no software (RODRIGUES et al., 2013), antes que o mesmo seja disponibilizado para uso.

A Engenharia de Software tem sugerido uma série de testes, metodologias e modelos de desenvolvimento para garantir sistemas de software entregues com maior robustez e segurança (SOMMERVILLE et al., 2003). Para garantir a segurança, experiências mostraram que a detecção de vulnerabilidades em software são mais eficazes quando feitas por engenheiros treinados e especializados no assunto. Por exemplo, testes e inspeções são guiados pelo conhecimento técnico ou mesmo intuição desses profissionais (MCGRAW, 2006). Além disso, concentrar empenhos em segurança desde os estágios iniciais do ciclo de desenvolvimento do software, diminui custos e esforço para correções. *Software Engineering Institute*[1] (SEI) demonstrou que 45% das vulnerabilidade foram reduzidas quando a segurança foi considerada durante todo o ciclo de desenvolvimento do software. Para comparação, correções de vulnerabilidades em estágios finais de desenvolvimento custaram cerca de $2.5 milhões ao modo que focar em segurança desde o começo custou $0.5 milhão.

Pesquisas defendem que o aumento da complexidade estrutural de um programa

[1] https://www.cert.org/cybersecurity-engineering/

está relacionado com o aumento da probabilidade de incidências de vulnerabilidades no mesmo (SHIN; WILLIAMS, 2008b). Complexidade estrutural consiste de um conjunto de características da estrutura de um programa, tais como tamanho, coesão e acoplamento. Várias pesquisas usam métricas estruturais de código para realizar predição de vulnerabilidades baseadas nessa argumentação (CHOWDHURY; ZULKERNINE, 2010; SHIN et al., 2011; ALENEZI; ABUNADI, 2015). Métricas estruturais de código são valores que são atribuídos às características estruturais do código. Elas são usadas de forma combinada com técnicas de aprendizagem de máquina para treinamento de modelos preditivos. O objetivo é encontrar padrões ocultos a partir de dados de vulnerabilidades já conhecidas para prever ocorrências de vulnerabilidades ainda não reveladas.

Entender e relacionar os *insights* dos inspetores de vulnerabilidades, como verificar artefatos que recebam entradas de usuários verificando e analisando cuidadosamente essas estradas e atribuir essas percepções a modelos preditivos é uma tarefa difícil e ainda não comprovada pela literatura atual. Para realizar tal tarefa, essas pesquisas apostam no uso de métricas de software baseadas em complexidade de código como discriminadoras de vulnerabilidades. A hipótese é que artefatos (arquivos, classes, módulos e funções) de maior complexidade estrutural induzem programadores a introduzir vulnerabilidades no código.

1.1 PROBLEMA E TRABALHOS RELACIONADOS

Apesar dessas pesquisas, a exploração de vulnerabilidades ainda é causa de grandes prejuízos financeiros, ainda mais com o crescente aumento no uso de tecnologias e a evolução da computação ubíqua. Isso acontece pelo fato de que essa hipótese não tem sido bem estudada em sua essência. Vários modelos de predição não são baseados em evidências científicas convincentes que atestem a relação de causa e efeito entre complexidade estrutural e vulnerabilidades em programas. Mais além, mesmo que pesquisas nos últimos oito anos (SHIN et al., 2011; CHOWDHURY; ZULKERNINE, 2010; SHIN; WILLIAMS, 2013; SMITH; SHIN; WILLIAMS, 2008; SHIN; WILLIAMS, 2008a; SHIN; WILLIAMS, 2011; SHIN et al., 2011) tenham proposto o uso de métricas de software para realizar predição de vulnerabilidades, seus resultados ainda não são satisfatórios. Isso mostra que essas pesquisas precisam ser melhoradas e não na área de mineração de dados, mas em uma fase anterior, na escolha e seleção de *features*.

Ademais, é possível citar outras limitações dos trabalhos relacionados na literatura:

- Hipóteses formuladas de maneira subjetiva, defendendo a relação de complexidade e vulnerabilidades com o seguinte argumento: complexidade é inimiga da segurança (MCGRAW, 2006), ao qual ficaria mais difícil para o desenvolvedor entender o código complexo e assim, deixa-lo mais propenso a introduzir uma vulnerabilidade. Entretanto, os trabalhos existentes não especificam de que maneira essas métricas se relacionam

diretamente com a presença da vulnerabilidade. Mesmo que evidências estatísticas mostrem que há uma relação em potencial, resultados de estudos conduzidos por diferentes pesquisadores têm sido contraditórios. Até o momento, não parece haver consistência na relação entre características estruturais específicas e tipos de vulnerabilidades na medida que diferentes amostras são utilizadas.

- Os trabalhos existente não especificam e estudam tipos específicos de vulnerabilidades. Simplesmente relacionam alta complexidade estrutural de código ou outras métricas a todo e qualquer tipo de vulnerabilidade. Porém, é sabido que as vulnerabilidades podem ter caráter diferentes. Por exemplo, uma vulnerabilidade pode ser causada pela invalidade de dados de entrada e outra pela ausência de autenticação. Ademais, cada vulnerabilidade pode ser materializada em um programa como poucas ou até mesmo uma linha de código. Também podem se comportar de diversas formas.

- Estudos existentes fazem uso de amostras pouco representativas, pois estudam uma pequena parcela do histórico de um projeto, frequentemente ignorando versões onde as vulnerabilidades foram introduzidas. Ademais, tais amostras não incluem projetos de diversos domínios diferentes, com características e tamanhos diferentes. Será que um projeto pequeno com poucos artefatos complexos pode ter quantidade de vulnerabilidades considerável e não serem previstas pelos modelos preditivos criados? Os artigos não respondem a esse questionamento.

- As vulnerabilidades podem estar associadas com uma única função (ou método), ou talvez até com artefatos menores como instruções. Porém, a granularidade que esses estudos existentes investigam é a nível de arquivos. Arquivos podem conter milhares de linhas de código, que combinadas com resultados incorretos dos modelos preditivos, podem retornar quantidade insatisfatória de artefatos para realização de inspeção manual.

Diante de todos esses problemas, é necessária, a priori, entender como métricas estruturais de código podem discriminar vulnerabilidades. É preciso entender o porquê da relação entre características estruturais e tipos específicos de vulnerabilidades. Análises estatísticas não são suficientes e consistem apenas um dos passos necessários.

1.2 RELEVÂNCIA

A principal importância deste trabalho é direcionar novas pesquisas a um rumo mais conciso e equilibrado, da forma que as investigações tenham maior fundamentação sobre a relação complexidade estrutural e vulnerabilidades de software, isso a partir de observações de casos reais e aplicação de inteligência artificial para comprovar e intensificar essa relação.

O conjunto de dados será composto por sete projetos com diferentes contextos, linguagens e tamanhos, representando assim um *dataset* representativo e com diversas informações sobre vulnerabilidades corrigidas nesses projetos. Esse conjunto de dados será disponibilizado e tudo isso servirá como suporte para dar condições a estudos futuros e aprimorar análises relacionadas a vulnerabilidades.

É observável que esses dados sofrem com problemas comuns (balanceamento por exemplo) e que a aplicabilidade de técnicas está sendo feita com metodologias desajustadas e mal combinadas. A ideia aqui é explorar algumas técnicas e métodos como sugestão para encontrar e aconselhar um ponto de maior precisão na predição de vulnerabilidades.

1.3 OBJETIVO

Tendo em vista a necessidade de se evitar a remanescência de vulnerabilidades no código fonte, é importante entender os fatores que induzem desenvolvedores a introduzir vulnerabilidades em programas. Um dos fatores mais citados na literatura da Engenharia de Software é a complexidade estrutural do código. Porém, embora a literatura especule que exista um relacionamento entre complexidade estrutural e introdução de vulnerabilidades em programas, ainda pouco se sabe sobre este fenômeno.

O objetivo, então, deste trabalho é entender como essas métricas realmente podem discriminar as vulnerabilidades através do uso de uma série de métodos científicos complementares. Suspeita-se que alguns tipos de vulnerabilidades não serão representadas por essas métricas de software. Assim como já foi dito, as vulnerabilidades existem por diversos motivos e são de diferentes tipos, por isso em alguns momentos, são sugeridos direcionamentos específicos para pesquisas futuras sempre observando essa posição. Mais adiante, é descrito a definição de alguns tipos de vulnerabilidades mais frequentes em relatórios de vulnerabilidade de projetos *open source*.

Desta forma, foram selecionados dez casos reais desses repositórios de vulnerabilidades e estudado como suas métricas se relacionam com funções vulneráveis. A ideia principal é investigar esses casos e a partir das observações encontrar *insights* que podem nos ajudar a direcionar metodologias.

Outro objetivo é melhorar novas investigações a partir das conclusões e *findings* juntamente com um conjunto de dados representativos sobre vulnerabilidades corrigidas em projetos *open source*. Dessa forma é possível possibilitar a exploração de várias configurações de modelos de aprendizagem como validação cruzada, separação por tipos de vulnerabilidades, técnicas de redução de dimensionalidade, escolha dos melhores atributos, técnicas de balanceamento, entre outros.

1.4 ESTRUTURA DA DISSERTAÇÃO

Este trabalho está dividido em mais 9 capítulos descritos a seguir:

- No **Capítulo 2** é discorrido sobre conceitos fundamentais e discutido sobre dez exemplos reais de vulnerabilidades em funções.
- No **Capítulo 3** será descrito com detalhes o processo de coleta, extração e armazenamento da base de dados de vulnerabilidade.
- O **Capítulo 4** apresenta um estudo empírico sobre a base de dados construída baseado em testes estatísticos, mostrando indícios de que existe alguma relação entre vulnerabilidades e métricas de código.
- No **Capítulo 5** é realizada uma análise comparativa dos modelos de predição atuais existentes na literatura que se baseiam em métricas de software no escopo de complexidade dos arquivos.
- No **Capítulo 6** é relatado, a partir de dados estatísticos, as melhores métricas de software quando selecionadas para discriminar vulnerabilidades.
- No **Capítulo 7** mostra-se algumas limitações e ameaças à validade do trabalho.
- No **Capítulo 8** é discutido sobre os trabalhos relacionados dividindo-os em categorias que contribuíram para o avanço da dissertação.
- No **Capítulo 9** é apresentado as dificuldades e oportunidades decorridas da realização deste trabalho.
- No **Capítulo 10** é apresentado as conclusões retiradas de todo o trabalho, mostrando assim os trabalhos futuros, os trabalhos em progresso e as considerações finais.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo será discutido conceitos e aspectos que levaram a intuição da relação complexidade estrutural e vulnerabilidades de software. Será mostrado vários conceitos importantes como definições de tipos de vulnerabilidades, as métricas de software que serão utilizadas, exemplos motivacionais e como eles se relacionam com os outros pontos.

2.1 VULNERABILIDADES CONHECIDAS

Empresas e comunidades abertas se preocupam com investimentos em segurança de software, investigando gastos, perdas e vulnerabilidades mais frequentes (LEWIS; BAKER, 2013; OWASP, 2013). Essa preocupação é refletida por projetos *open souce* que utilizam repositórios para reportar vulnerabilidades encontradas, e assim corrigi-las.

Em um estudo inicial, verificam-se as seguintes vulnerabilidades e sua respectiva definição. Elas foram destacadas pela frequência com que foram reportadas em estudos e relatórios de vulnerabilidades.

- *Denial of service* (DoS): Atacantes podem **consumir recursos** ou impedir o funcionamento de aplicações web de uma forma que os usuários legítimos não conseguem mais acessar ou usar a aplicação.
- *Code Execution*: O software incorpora em seu código, todo ou parte de um segmento de instruções utilizando **entradas influenciadas externamente** provindas de um componente de entrada. Porém, esse componente não neutraliza ou neutraliza incorretamente elementos especiais que podem modificar a interpretação e comprometer o comportamento do sistema.
- *Overflows*: é uma anomalia onde um programa, ao escrever dados em um *buffer*, ultrapassa os limites do *buffer* e sobrescreve em memória adjacente. **Violando limites de alocação de memória**, podendo gerar erros de acesso à memória e parada do sistema.
- *Directory Traversal*: Ao utilizar elementos externos para criar um caminho até um arquivo, o software não trata esses elementos, podendo levar a um acesso fora da área restrita de acesso.
- *Bypass*: O software **não executa autenticação** de forma correta, permitindo que um usuário malicioso pratique ações sem a devida autenticação e com permissões não adequadas.

Vendo isso, é perceptível que algumas vulnerabilidades estão diretamente relacionadas a certos procedimentos, tais como autenticação, validação de dados de entrada, violações do limite de alocação de memória e consumo de recursos. Sendo assim, o objetivo é identificar a relação de falhas nesses processos que podem levar a vulnerabilidades e métricas estruturais de código.

2.2 MÉTRICAS DE SOFTWARE

Métricas de software são mecanismos que permitem mensurar algo em um artefato de software. Elas podem representar acoplamento, coesão, complexidade e diversos outros atributos. Trabalhos já citados como (SHIN et al., 2011; CHOWDHURY; ZULKERNINE, 2010; SHIN; WILLIAMS, 2013; SMITH; SHIN; WILLIAMS, 2008; SHIN; WILLIAMS, 2008a; SHIN; WILLIAMS, 2011) utilizaram métricas de software para realizar a predição de vulnerabilidades. Apesar de não especificar explicitamente a relação entre essas métricas e as vulnerabilidades, eles se defendem com a hipótese de que quanto mais complexo o código, mais fácil para o desenvolvedor deixar uma brecha de segurança.

As métricas de software podem ser de vários tipos. Neste estudo, o foco foi em métricas de caráter estrutural, representando volume de linhas, contagem de instruções, contagem de *tokens*, grafo de controle de fluxo, comentários e chamadas externas.

Métricas de volume estão relacionadas a quantidade de linhas de algum artefato de código, seja ele função, classe ou arquivo. Esse tipo de métrica influencia na legibilidade do código, visto que artefatos com grandes quantidades de código são mais difíceis de manter e entender. Porém, quando se remete a influência dessa características a vulnerabilidades, nota-se que essa premissa é aplicável a todos os defeitos que podem aparecer no código, não só a vulnerabilidades. Alguns exemplos dessas métricas e sua respectiva definição:

- *CountLine*: é a quantidade total de linhas ocupada pela função.
- *CountLineCode*: é uma métrica de contagem básica que representa a quantidade de linhas que contém código fonte.
- *CountLineCodeExe*: conta quantas linhas de código são utilizadas no escopo (corpo) da função.
- *CountLineCodeDelc*: conta quantas linhas de código são utilizadas para declarar a função.
- *CountLineBlank*: quantidade de linhas em branco.

Métricas de contagem de *tokens* são métricas que contam a quantidade de símbolos no escopo de uma função. Esse tipo de métrica foi criada para representar a complexidade do

código em termos de operadores da linguagem, de modo que quanto mais o uso de *tokens* de operações específicas (seleção, agrupamento de sentença, repetição), mais complexo o código, representando ilegibilidade do código.

A Tabela 1 mostra algumas métricas de software e seus *tokens* contabilizados. Esse tipo de métrica de contagem de *tokens* não parece tão discriminativa ao ponto de detectar uma vulnerabilidade, porém em casos específicos, tais como violações de valores limítrofes - quando o desenvolvedor não fecha o escopo da seleção e deixa que dados não esperados comprometam o comportamento do sistema, ou estruturas de controle mal implementadas - quando o desenvolvedor implementa um *switch case* e não fecha o laço quando deveria ao entrar em cases, podem servir de padrões que evidenciem certo tipo de vulnerabilidade.

Tabela 1 – Métricas de software e seus respectivos *tokens* de contagem.

Métrica	**AND**	**OR**	**CASE**	**CATCH**	**DO**	**FOR**	**IF**	**?**	**WHILE**	**;**	**SWITCH**
Cyclomatic			X	X	X	X	X	X	X		
CyclomaticModified				X	X	X	X	X	X		X
CyclomaticStrict	X	X	X	X	X	X	X	X	X		
CountSemicolon										X	

Fonte: Adaptada de (SCITOOLS, 2016).

Além dessas, existem métricas de contagem de *tokens* que medem o aninhamento das funções. Essa métrica reflete o quão aninhado estão as estruturas de controle e seleção da função, representando uma medida de profundidade da função. Esse retorno pode ser interessante em casos de vulnerabilidades que apresentam falhas no fechamento de escopo. A hipótese aqui é que quanto mais aninhadas estão as instruções, mais fácil o desenvolvedor poderá deixar de fechar o escopo de estruturas de controle que podem acarretar em atividades inesperadas do sistema. A métrica é descrita logo a seguir:

- *MaxNesting*: Contagem de aninhamento da função, basicamente o parse mantém o controle do aninhamento incrementando a métrica quando entra em escopos com **DO**, **ELSE**, **FOR**, **IF**, **WHILE** e **SWITCH** decrementando quando sai desses escopos. O maior valor registrado é o valor do *MaxNesting* .

Métricas sobre o grafo de controle de fluxo é uma abordagem um pouco mais complexa que as demais métricas já apresentadas até aqui. Ferramentas de detecção de vulnerabilidades já foram implementadas utilizando essa abordagem juntamente com análise de fluxo de dados (SAMPAIO; GARCIA, 2016). A estratégia aqui é, ao invés de contar linhas ou *tokens*, contar nós em um grafo pelo qual representa o fluxo de controle do algoritmo. A estrutura do fluxo captura parcialmente o comportamento do sistema. Dentre as métricas, destacam-se:

- *Knots*: é a quantidade de nós no gráfico, essa métrica destaca os momentos na qual o código pode seguir caminhos distintos, representando controladores de fluxo.
- *Essential*: é a quantidade de filhos extra, pelo qual cada nó pode apresentar. Nesse caso, ignoram-se nós com apenas uma ramificação e conta a quantidade de ramificações (arestas) menos um. Praticamente é uma métrica semelhante ao *Knots*, mas adaptada, para ignorar estruturas de controle que nunca são acionadas.
- *CountPath*: métrica que conta a quantidade de caminhos possível em um código. Essa métrica é a responsável pela quantidade de possibilidades que um código pode executar. Essa métrica é mais conhecida como *McCabe's complexity*, representando a complexidade ciclomática do código.

Métricas baseadas em contagem de instruções são métricas diferentes das métricas de linhas de código. Um parser indica se a instrução é declarativa, de execução ou vazia. E conta sua incidência. Esse tipo de métrica se assemelha a métricas de contagem de linha. Entretanto, ao invés de contar linhas, conta instruções. Como no exemplo a seguir e suas respectivas métricas:

Code 2.1 – Demonstração de código

```
1 for (int i = 0;              \\ declarativa
2     i < 10;                  \\ executável
3     i++)                     \\ executável
4 ;                            \\ vazia
```

- *CountStmtDecl*: conta instruções declarativas;
- *CountStmtExe*: conta instruções de execução;
- *CountStmtEmpty*: conta instruções vazias;
- *CountStmt* = *CountStmtDecl* + *CountStmtExe* + *CountStmtEmpty*.

Essas métricas podem nos dar informações mais concisas que as de linhas de código, por que uma instrução pode estar em mais de uma linha de código. Entretanto, não muda o fato que são apenas métricas de contagem, que medem o volume de instruções correspondendo a complexidade estrutural de modo geral.

Algumas outras métricas também podem ser bem exploradas. Métricas de chamadas internas são importantes para verificar o quão essas funções atuam sobre dados externos, contabilizando chamadas de subprogramas, leitura e modificação de variáveis globais. Vulnerabilidades que têm relação com validação de entrada de dados (*code execution*, *injections*) poderiam ter relação com essas métricas, ao ponto que mediria o quão uma

função está manipulando dados externos, podendo esta ser um foco para realizar inspeções verificando se o desenvolvedor tratou corretamente todas as estradas de dados. As métricas são descritas a seguir:

- *CountInput*: É o número de chamadas de funções com finalidade apenas de leitura de dados (sem modificações) mais o número de variáveis globais lidas. É interessante notar que algumas vulnerabilidades são causadas pela falta de verificação ou validação de dados de entradas. Essas métricas podem representar a quantidade de dados que a função recebe do ambiente exterior;
- *CountOutput*: Quantidade de chamadas de subprogramas ou variáveis globais pelo qual pode ser realizado alguma modificação de dados (SETs). Essa métrica é importante, pois modifica dados de entrada da função que podem comprometer o sistema.

Métricas de linhas comentadas são métricas que contabilizam linhas que apresentam comentários. Comentários podem determinar ações dos desenvolvedores, entretanto, tais ações podem ser específicas e diferentes. Desenvolvedores podem comentar códigos mais complexos, explicando o funcionamento do código, tanto quanto podem comentar apenas informações adicionais necessárias que não tem nada a ver com complexidade, talvez comentem as funções mais importantes, independe de complexidade estrutural. Então esse estudo é realizado verificando se a partir desses pontos, as linhas comentadas tem alguma relação com vulnerabilidades. Essas são as métricas no contexto de comentário de código:

- *RatioCommentToCode*: taxa de linhas comentadas;
- *CountLineComment*: número de linhas que contém comentários.

Desta forma, diversas métricas de software são destacadas abordando contextos diferentes. Ficando mais fácil assim, associar tais métricas com determinados tipos de vulnerabilidades e descobrir se as práticas de desenvolvedores associadas a essas métricas estão relacionadas, de alguma forma, a certos tipos de vulnerabilidades.

2.3 EXEMPLOS MOTIVACIONAIS

Para exemplificar algumas intuições, foi separado alguns exemplos aleatórios da nossa base de dados a fim de discutir como as mudanças ajustaram o código e a relação das funções associadas a essas mudanças e às métricas de software.

A metodologia de seleção dos exemplos ocorreu da seguinte forma: foram selecionados 50 casos aleatórios da base de dados e dentre eles foram escolhidos os 10 mais interessantes. Dentre esses 10 casos estão exemplos mais frequentes e alguns que podem ser situações controversas para o entendimento da nossa hipótese.

A fim de analisar esses exemplos, as métricas que mais se afastam da mediana e média de toda a amostra foram investigadas. Essa seleção de métricas inicias é só um filtro das métricas pelo qual se observa a correlação entre as métricas e as vulnerabilidades em um primeiro momento. A seleção foi da seguinte forma: os dez exemplos foram separados e todas as métricas verificadas separando aquelas que mais mostram discrepância e valores fora do comum em relação a média e mediana. A Figura 1 mostra os valores de média e mediana das métricas selecionadas para discussão. Os valores dessas métricas associada a cada método será comparado aos valores da média e mediana da amostra do projeto.

Figura 1 – Média e mediana das amostras

Project	Métricas	CountLine	CountSemicolon	CountStmt	MaxNesting	Cyclomatic	CyclomaticStrict	Knots	Essential	CountPath	RatioCommentToCode	CountInput	CountOutput
Mozilla	média	27	12	16	1	5	6	18	3	2508239	0,07	15	7
	mediana	9	4	5	1	2	2	0	1	2	0,00	4	4
Kernel	média	36	21	26	2	9	10	8	3	4355611	0,08	52	8
	mediana	20	10	13	1	4	4	1	1	3	0,00	7	5
Xen	média	43	24	31	1	10	11	71	5	8484250	0,06	14	9
	mediana	18	8	11	1	3	3	1	1	3	0,00	6	5
Htpd	média	59	20	27	2	8	10	11	4	8717657	0,13	11	9
	mediana	26	10	12	1	4	4	1	1	3	0,03	7	5
Glibc	média	51	34	43	2	20	27	19	4	21192489	0,10	12	6
	mediana	21	10	13	1	4	4	1	1	4	0,00	4	4
Tomcat	média	18	6	9	1	3	3	2	1	1130619	1,00	8	5
	mediana	8	2	3	0	1	1	0	1	1	0,14	3	2
Derby	média	30	8	11	1	3	3	3	1	1912593	1,23	15	5
	mediana	13	2	4	0	1	1	0	1	1	0,38	4	3

Fonte: elaborada pelo autor.

A primeira observação retirada desse resultado é que o valor de média e mediana é muito distinto, isso mostra que as amostras apresentam valores extremamente discrepantes e que a própria média pode não representar a distribuição global das amostras, levando a considerar a mediana também como medida de tendência central.

Majoritariamente, a média foi maior que a mediana, isso mostra que as amostras apresentam um número maior de funções com valores baixos (menos complexos) em comparação com o número de funções mais complexas e que essas funções complexas apresentam valores muito discrepantes, o que pode ser um indicativo interessantes para a investigação dos exemplos que serão discorridos adiante.

A Figura 2 apresenta um resumo dos exemplos e os valores das métricas dos métodos vulneráveis, pelo qual destaca-se em vermelho os valores que superam a média e de amarelo os valores que superam a mediana da amostra do seu respectivo projeto.

O primeiro exemplo é retirado da amostra do Mozilla e caracteriza uma corrupção de memória que causa a perda dos serviços normais do sistema. Nesse exemplo, duas funções em diferentes arquivos precisaram ser alteradas para conter a vulnerabilidade: as funções *ConstructPath* e *PaintSVG*.

As Figuras 3 e 4 representam as funções com as mudanças para correção da vulnerabilidade, destacado em vermelho as linhas que foram removidas e de verde as linhas que foram adicionadas para que a vulnerabilidade deixasse de existir.

A vulnerabilidade acontece porque não era esperado que variáveis de altura e largura pudessem ser negativos, e de alguma forma os usuários conseguiram fazer essa alteração em memória, e quando isso acontecia, a vulnerabilidade era efetivada causando a perda do serviço. Nesse caso específico, nota-se que o único ponto de fraqueza foi o desenvolvedor não ter fechado o escopo completamente, mas não é possível afirmar com clareza que esse deslize foi causado pela complexidade estrutural da função, mas essa complexidade estava presente na função mais modificada: *PaintSVG*. A solução foi a adição de um novo condicional, limitando o fluxo do algoritmo quando os valores forem negativos e tratando isso de forma correta.

Observando as métricas das funções, a função *ConstructPath (gfxContext*)* é uma função pequena, 12 linhas de código, não complexa e com os valores de métricas bem inferiores.

Figura 2 – Métricas das amostras

Projeto	#	ID Dataset	Função/Método	Vulnerabilidade(s)	CountLine	CountSemicolon	CountStmt	MaxNesting	Cyclomatic	CyclomaticStrict	Knots	Essential	CountPath	RatioCommentToCode	CountInput	CountOutput
Mozilla	1	vuln464mzl	PaintSVG(...)	DenialOfService e Memorycorruption	71	27	40	3	14	14	1	1	651	0,06	5	23
		vuln464mzl	ConstructPath(gfxContext*)	DenialOfService e Memorycorruption	12	4	5	1	2	3	1	1	2	0	1	3
	2	vuln956mzl	OnRedirectStateChange(...)	Undefined	46	14	20	2	8	8	3	4	21	0,29	6	10
Kernel	3	ker650v	dns_resolve_server_name_to_ip(...)	Undefined	73	44	64	7	31	32	9	6	712	0,03	10	13
	4	ker397v	main()	Undefined	230	110	139	3	28	28	70	5	12704	0,16	29	40
Xen HV	5	xen90v	hvm_domain_relinquish_resources(...)	DenialOfService	23	11	13	1	3	4	0	1	4	0,06	12	8
		xen90v	hvm_domain_destroy(structdomain*)	DenialOfService	8	5	5	0	1	1	0	1	1	0	3	5
	6	xen7v	get_page_from_gfn(...)	DenialOfService	14	5	7	1	4	4	1	1	3	0,09	6	5
Httpd	7	httpd18v	ap_cleanup_scoreboard()	DenialOfService	15	6	8	1	3	3	2	1	3	0	3	4
		httpd18v	ap_create_scoreboard(...)	DenialOfService	47	16	18	2	3	3	1	1	3	0,04	5	13
Tomcat	8	tomcat126v	ProcessSendfile(...)	Information Disclosure	95	35	54	5	21	22	15	6	485	0,12	20	27
	9	tomcat73v	CreateServer(...)	Code Execution	53	19	24	1	6	6	7	3	9	0,04	17	9
		tomcat73v	lifecycleEvent(LifecycleEvent)	Code Execution	87	26	40	3	14	14	5	1	514	0,21	19	11
Derby	10	derby24v	getTraceDirectory(Properties)	Security Bug	16	4	6	1	2	2	1	1	2	1	5	3
		derby24v	run()	Security Bug	3	1	2	0	1	1	0	1	1	0	1	2
		derby24v	readSystemProperty(String)	Security Bug	23	2	4	0	1	1	0	1	1	1,56	2	2

Maior que a mediana
Maior que a média

Fonte: elaborada pelo autor.

Figura 3 – Vulnerabilidade em ConstructPath

```
void
nsSVGImageElement::ConstructPath(gfxContext *aCtx)
{
  float x, y, width, height;

  GetAnimatedLengthValues(&x, &y, &width, &height, nsnull);

-  if (width == 0 || height == 0)
+  if (width <= 0 || height <= 0)
     return;

  aCtx->Rectangle(gfxRect(x, y, width, height));
}
```

Fonte: elaborada pelo autor.

Figura 4 – Vulnerabilidade em PaintSVG

```
//------------------------------------------------------------------------
// nsISVGChildFrame methods:
NS_IMETHODIMP
nsSVGImageFrame::PaintSVG(nsSVGRenderState *aContext, nsRect *aDirtyRect)
{
  (...)

+  float x, y, width, height;
+  nsSVGElement *element = static_cast<nsSVGElement*>(mContent);
+  element->GetAnimatedLengthValues(&x, &y, &width, &height, nsnull);
+  if (width <= 0 || height <= 0)
+    return NS_OK;
+
  (...)
      if (ctm) {
-       float x, y, width, height;
-       nsSVGElement *element = static_cast<nsSVGElement*>(mContent);
-       element->GetAnimatedLengthValues(&x, &y, &width, &height, nsnull);

        nsSVGUtils::SetClipRect(gfx, ctm, x, y, width, height);
      }

  (...)
  return rv;
}
```

Fonte: elaborada pelo autor.

Por outro lado a função *PaintSVG(nsSVGRenderState*, nsRect*)* apresenta *CountPath* com valor 651, refletindo uma complexidade fora do comum, e com 71 linhas de código, além de ter suas métricas com valores maiores que média e medianas.

Desta forma, uma função que precisou ser reparada para conter a vulnerabilidade por si só, não apresenta complexidade em suas métricas. Para ir mais fundo, as métricas de arquivo são avaliadas, que apesar de não ser o foco deste trabalho, é interessante investigar.

Observando essa outra perspectiva, as métricas dos arquivos, nota-se alguns valores relativamente altos, o arquivo *nsSVGImageElement.cpp*, que contém a função *PaintSVG*, apresenta 366 linhas de código e *CountStmt* equivalente a 252, outra métrica que se destacou foi *RatioCommenttoCode* com valor de 0.41, o que é possível afirmar que pouco menor que a metade das linhas do arquivo são linhas com comentários. Já o outro arquivo, *nsSVGImageFrame.cpp*, que contém a função *ConstructPath*, apresenta 430 linhas de código mas algumas métricas se destacaram, *CountStmt* com valor 213, e *CountPath* com 651 o que demostra uma complexidade além da capacidade normal que um desenvolvedor tem para avaliar o código.

No exemplo 2, apresentado na Figura 5, retirado da amostra do Mozilla com métricas acima da média, a mudança para a correção da vulnerabilidade foi apenas a adição de uma estrutura condicional.

A métrica que mais chama atenção nesse exemplo 2 é a *RatioCommentToCode* que apresenta 29% de linhas de comentários em relação as linhas de código, mas as demais apesar de serem superiores à média, não apresentam valores discrepantes. Essa observação indica

Figura 5 – Vulnerabilidade em OnRedirectStateChange

```
 void
 nsDocShell::OnRedirectStateChange(nsIChannel* aOldChannel,
                                   nsIChannel* aNewChannel,
                                   PRUint32 aRedirectFlags,
                                   PRUint32 aStateFlags)
 {
     (...)

     // check if the new load should go through the application cache.
     nsCOMPtr<nsIApplicationCacheChannel> appCacheChannel =
         do_QueryInterface(aNewChannel);
     if (appCacheChannel) {
         nsCOMPtr<nsIURI> newURI;
         aNewChannel->GetURI(getter_AddRefs(newURI));
         appCacheChannel->SetChooseApplicationCache(ShouldCheckAppCache(newURI));
     }
+
+    if (!(aRedirectFlags & nsIChannelEventSink::REDIRECT_INTERNAL) &&
+        mLoadType & (LOAD_CMD_RELOAD | LOAD_CMD_HISTORY)) {
+        mLoadType = LOAD_NORMAL_REPLACE;
+        SetHistoryEntry(&mLSHE, nsnull);
+    }
 }
```

Fonte: elaborada pelo autor.

que não se pode verificar média e mediana como valores que discriminam vulnerabilidade, ou em outras palavras, não se pode afirmar que uma função que tenha suas métricas maiores que a média implica que ela seja forte indicação de ter vulnerabilidades. Assim, é interessante ir um pouco mais fundo, a nível de arquivos.

Quando o nível de granularidade é aumentado e a observação se dá a nível de arquivo, percebe-se que esse arquivo é grande, apresentando 11479 linhas de código, juntamente com muitas instruções onde *CountStmt* é equivalente a 6.058 instruções, contrapondo-se a métrica *CountPath* que apresenta valor baixo. Esse é um exemplo que ao ver métricas apenas de funções, ou somente as métricas selecionadas neste trabalho, fica difícil associar essas características a vulnerabilidades nesse nível de funções.

Indo mais adiante, o exemplo 3 apresentado na Figura 6 nos mostra uma função que era responsável pela vulnerabilidade, mas nesse caso houve alterações em três arquivos, além da adição de outras funções, a vulnerabilidade foi encontrada na função *dns_resolve _server_name_to_ip (constchar*, char**)* que apresenta valores de algumas métricas muito altas, por exemplo *CountPath* 712 e *CountStmt* 64. Mas a complexidade do problema vem, não apenas da função, mas também de *structs* e outros artefatos com granularidade maior (mudanças na configuração do arquivo, variáveis, includes).

Como quarto exemplo, retirado da amostra do projeto Kernel, está destacado a função *main*, função que apresenta 230 linhas de código com valores de outras métricas acima da média, por exemplo *CountStmt* igual a 139, *CountOutput* igual a 40, *Knots* igual a 70 e CountPath com valor surpreendente de 12704. Notavelmente, é uma função complexa e que tal complexidade pôde ter influenciado em algum aspecto de legibilidade e compreensão que

Figura 6 – Vulnerabilidade em dns_resolve_server_name_to_ip

```
 int
 dns_resolve_server_name_to_ip(const char *unc, char **ip_addr)
 {
+     const struct cred *saved_cred;
      int rc = -EAGAIN;
      (...)

+     saved_cred = override_creds(dns_resolver_cache);
      rkey = request_key(&key_type_dns_resolver, name, "");
+     revert_creds(saved_cred);
      if (!IS_ERR(rkey)) {
+             if (!(rkey->perm & KEY_USR_VIEW)) {
+                     down_read(&rkey->sem);
+                     rkey->perm |= KEY_USR_VIEW;
+                     up_read(&rkey->sem);
+             }

              len = rkey->type_data.x[0];
              data = rkey->payload.data;
      (...)
      return rc;
 }
```

Fonte: elaborada pelo autor.

permitiu tal vulnerabilidade ser inserida. Essa função está sendo representada pela Figura 7.

Observando as correções, houve aumento maior do escopo de uma estrutura de seleção (if) e a troca do uso de uma função, ademais não vê-se mudanças (dentro do nosso escopo) que ajudem a entender tal código pela sua complexidade estrutural.

Figura 7 – Vulnerabilidade em *main*

```
 int main(void)
 {
      (...)

      while (1) {
+             struct sockaddr *addr_p = (struct sockaddr *) &addr;
+             socklen_t addr_l = sizeof(addr);
              pfd.events = POLLIN;
              pfd.revents = 0;
              poll(&pfd, 1, -1);

-             len = recv(fd, kvp_recv_buffer, sizeof(kvp_recv_buffer), 0);
+             len = recvfrom(fd, kvp_recv_buffer, sizeof(kvp_recv_buffer), 0,
+                            addr_p, &addr_l);

-             if (len < 0) {
-                     syslog(LOG_ERR, "recv failed; error:%d", len);
+             if (len < 0 || addr.nl_pid) {
+                     syslog(LOG_ERR, "recvfrom failed; pid:%u error:%d %s",
+                                     addr.nl_pid, errno, strerror(errno));
                      close(fd);
                      return -1;
              }
              (...)
 }
```

Fonte: elaborada pelo autor.

No exemplo 5, retirado da amostra do Xen Hipervisor, apresenta a remoção de linhas em uma função e a adição em outra função (demonstrado pelas Figuras 8 e 9). Pela descrição

da vulnerabilidade no site do *Xen Security Advisories*[1], existem dados de memória que são liberados prematuramente, possibilitando usuários novos a capturarem tais dados. A solução foi colocar a liberação desses dados em um momento adequado.

Figura 8 – Vulnerabilidade em hvm_domain_relinquish_resources

```
 void hvm_domain_relinquish_resources(struct domain *d)
 {
-    xfree(d->arch.hvm_domain.io_handler);
-    xfree(d->arch.hvm_domain.params);
-
     if ( is_pvh_domain(d) )
         Return;

     if ( hvm_funcs.nhvm_domain_relinquish_resources )
         hvm_funcs.nhvm_domain_relinquish_resources(d);

     (...)
 }
```

Fonte: elaborada pelo autor.

Figura 9 – Vulnerabilidade em hvm_domain_destroy

```
 void hvm_domain_destroy(struct domain *d)
 {
+    xfree(d->arch.hvm_domain.io_handler);
+    xfree(d->arch.hvm_domain.params);
+
     hvm_destroy_cacheattr_region_list(d);

     if ( is_pvh_domain(d) )
         return;

     hvm_funcs.domain_destroy(d);
     rtc_deinit(d);
     stdvga_deinit(d);
     vioapic_deinit(d);
 }
```

Fonte: elaborada pelo autor.

Observando as métricas dessas funções, percebe-se que elas são baixas, e que se avaliada apenas com as métricas de função, seria mais um falso negativo. Entretanto, quando o nível de granularidade é mudado para métricas de arquivo, percebe-se que o arquivo apresenta uma complexidade considerável, com *CountLine* superior a 4.000 linhas, *CountPath* igual a 12.887.095, *CountInput* e *CountOutput* superior a 300 e *CountStmt* igual a 2.700. Esse é mais um exemplo que indica que é preciso usar as métricas de função relacionadas com métricas de arquivos.

O exemplo 6 foi retirada da amostra do projeto Xen, nele é apresentado um *DenialOfService* que ocorre em um retorno impróprio de uma função. Basicamente para corrigi-lo uma função de validação que antes era ignorada foi adicionada a um condicional de retorno da função. A função *get_page_from_gfn(structdomain*, unsignedlong, p2m_type_t*, p2m_query_t)* tem todos os valores de métricas baixos, ela é representada pela Figura 10.

[1] http://xenbits.xen.org/xsa/advisory-116.html

Esse exemplo é um exemplo básico de uma função pequena e não complexa pelo qual se apresentou com vulnerabilidades sendo assim um falso-negativo. Visualmente falando, com essas métricas fica improvável associa-las a essa vulnerabilidade. Um ponto interessante em destaque é que o tipo de vulnerabilidade foi catalogado como um efeito do ataque (*DenialOfService*) e não um meio de ataque, podendo este ser uma característica que também possa estar influenciando os resultados. Será que apenas certos tipos de vulnerabilidades estariam associados à complexidade de código?

Figura 10 – Vulnerabilidade em get_page_from_gfn

```
 static inline struct page_info *get_page_from_gfn(
     struct domain *d, unsigned long gfn, p2m_type_t *t, p2m_query_t q)
 {
     struct page_info *page;

     if ( paging_mode_translate(d) )
         return get_page_from_gfn_p2m(d, p2m_get_hostp2m(d), gfn, t, NULL, q);

     /* Non-translated guests see 1-1 RAM mappings everywhere */
     if (t)
         *t = p2m_ram_rw;
     page = __mfn_to_page(gfn);
-    return get_page(page, d) ? page : NULL;
+    return mfn_valid(gfn) && get_page(page, d) ? page : NULL;
 }
```

Fonte: elaborada pelo autor.

Outro exemplo interessante foi retirado da amostra do httpd, em que uma mudança sutil foi realizada a fim de armazenar em uma variável do objeto manipulador ao invés de chamá-lo em outro objeto que poderia mais facilmente sofrer modificações. A vulnerabilidade do exemplo 7 consiste em um *Denial of Service* ativado ao modificar a tipagem de um determinado campo dentro de um segmento de memória compartilhada. A correção dessa vulnerabilidade pode ser vista pelas Figuras 11 e 12.

Figura 11 – Vulnerabilidade em ap_cleanup_scoreboard

```
 apr_status_t ap_cleanup_scoreboard(void *d)
 {
     if (ap_scoreboard_image == NULL) {
         return APR_SUCCESS;
     }
-    if (ap_scoreboard_image->global->sb_type == SB_SHARED) {
+    if (scoreboard_type == SB_SHARED) {
         ap_cleanup_shared_mem(NULL);
     }
     else {
         free(ap_scoreboard_image->global);
         free(ap_scoreboard_image);
         ap_scoreboard_image = NULL;
     }
     return APR_SUCCESS;
 }
```

Fonte: elaborada pelo autor.

Observando as métricas dessa função, não se vê complexidade refletidas em suas métricas, mas ao observar métricas de arquivos, nota-se alguns valores altos, como *CountInput*

Figura 12 – Vulnerabilidade em ap_create_scoreboard

```
int ap_create_scoreboard(apr_pool_t *p, ap_scoreboard_e sb_type)
{
    (...)

-    ap_scoreboard_image->global->sb_type = sb_type;
+    scoreboard_type = sb_type;
     ap_scoreboard_image->global->running_generation = 0;
     ap_scoreboard_image->global->restart_time = apr_time_now();

     apr_pool_cleanup_register(p, NULL, ap_cleanup_scoreboard, apr_pool_cleanup_null);

     return OK;
}
```

Fonte: elaborada pelo autor.

igual a 42 (maior que a média) e *CountLine* igual a 590. Novamente esse é um caso de *Denial of Service* que apresenta mais representatividade para previsão de vulnerabilidades usando métricas de arquivos.

O exemplo 8 retirado da amostra do projeto Apache Tomcat é apresentado pela Figura 13, mostra a adição de uma estrutura condicional para evitar que o processador seja adicionado ao cache mais de uma vez. Esse é um típico caso em que as métricas de função refletem alta complexidade do método. Esse método apresenta *CountPath* igual a 485, com 95 linhas de código e *CountStmt* igual a 54.

Figura 13 – Vulnerabilidade em *processSendFile*

```
public SendfileState processSendfile(SelectionKey sk, KeyAttachment attachment,
        boolean calledByProcessor) {
    NioChannel sc = null;

    (...)

    }catch ( IOException x ) {
        if ( log.isDebugEnabled() ) log.debug("Unable to complete sendfile request:", x);
-       cancelledKey(sk,SocketStatus.ERROR);
+       if (!calledByProcessor) {
+           cancelledKey(sk,SocketStatus.ERROR);
+       }
        return SendfileState.ERROR;
    }catch ( Throwable t ) {
        log.error("",t);
-       cancelledKey(sk, SocketStatus.ERROR);
+       if (!calledByProcessor) {
+           cancelledKey(sk, SocketStatus.ERROR);
+       }
        return SendfileState.ERROR;
    }
}
```

Fonte: elaborada pelo autor.

O exemplo 9 é um erro de configuração que se corrige configurando explicitamente os tipos de credenciais permitidos, algo esquecido ou que não foi dada tanta importância na hora da implementação. Essa fato pode ter ocorrido pela complexidade do método *lifecycleEvent(LifecycleEvent)*, representado pela Figura 15, que apresenta a métrica de complexidade *CountPath* com valor alto (514), além da métrica *RatioCommentToCode* que

apresenta um percentual de 21% de linhas de comentário em relação às linhas de código. O interessante desse caso é que existe ainda uma modificação pequena a ser feita em outra função (*createServer* Figura14) que não apresenta valores de métricas muito altas, apesar da maioria superar a média. Esse tipo de caso pode prejudicar a eficiência dos algoritmos de aprendizagem, pois uma função pequena, com modificação ínfima está sendo classificada como vulnerável pela metodologia.

Figura 14 – Vulnerabilidade em CreateServer

```
@Override
public void lifecycleEvent(LifecycleEvent event) {
    // When the server starts, configure JMX/RMI
    if (Lifecycle.START_EVENT.equals(event.getType())) {
        (...)
        // Force the use of local ports if required
        if (useLocalPorts) {
            registryCsf = new RmiClientLocalhostSocketFactory(registryCsf);
            serverCsf = new RmiClientLocalhostSocketFactory(serverCsf);
        }

+       env.put("jmx.remote.rmi.server.credential.types", new String[] {
+               String[].class.getName(),
+               String.class.getName() });
+
        // Populate the env properties used to create the server
        if (serverCsf != null) {
            env.put(RMIConnectorServer.RMI_CLIENT_SOCKET_FACTORY_ATTRIBUTE, serverCsf);
            env.put("com.sun.jndi.rmi.factory.socket", registryCsf);
        }
    (...)
}
```

Fonte: elaborada pelo autor.

Figura 15 – Vulnerabilidade em lifeCycleEvent

```
private JMXConnectorServer createServer(String serverName,
        String bindAddress, int theRmiRegistryPort, int theRmiServerPort,
        HashMap<String,Object> theEnv,
        RMIClientSocketFactory registryCsf, RMIServerSocketFactory registrySsf,
        RMIClientSocketFactory serverCsf, RMIServerSocketFactory serverSsf) {

    (...)

    RMIConnectorServer cs = null;
    try {
        RMIJRMPServerImpl server = new RMIJRMPServerImpl(
                rmiServerPortPlatform, serverCsf, serverSsf, theEnv);
        cs = new RMIConnectorServer(serviceUrl, theEnv, server,
                ManagementFactory.getPlatformMBeanServer());
        cs.start();
-       registry.bind("jmxrmi", server);
+       registry.bind("jmxrmi", server.toStub());
        log.info(sm.getString("jmxRemoteLifecycleListener.start",
                Integer.toString(theRmiRegistryPort),
                Integer.toString(theRmiServerPort), serverName));
    } catch (IOException e) {
    (...)
    return cs;
}
```

Fonte: elaborada pelo autor.

O exemplo 10 foi retirado da amostra do projeto Derby e a correção é apresentada na Figura 16. A vulnerabilidade consiste em uma exceção não tratada sendo que sua correção foi a adição de um *try catch* para capturar a exceção. Observando métricas de métodos, só está em destaque a métrica de linhas comentadas pois existe mais linhas referente a comentários que linhas de código propriamente ditas. Já partindo para métricas de arquivos, nota-se que

o arquivo tem alta complexidade estrutural, apresentando mais de 1000 linhas de código, *CountStmt* igual a 365 e outras métricas com valores maiores que a média.

Figura 16 – Vulnerabilidade em *readSystemProperty*

```
 private static String readSystemProperty(final String key) {
     //Using an anonymous class to read the system privilege because the
     //method java.security.AccessController.doPrivileged requires an
     //instance of a class(which implements java.security.PrivilegedAction).
     //Since readSystemProperty method is static, we can't simply pass "this"
     //to doPrivileged method and have ClientBaseDataSource implement
     //PrivilegedAction. To get around the static nature of method
     //readSystemProperty, have an anonymous class implement PrivilegeAction.
     //This class will read the system property in it's run method and
     //return the value to the caller.
     return (String )AccessController.doPrivileged(new java.security.PrivilegedAction(){
         public Object run(){
-        return System.getProperty(key);
+        try {
+            return System.getProperty(key);
+        } catch (SecurityException se) {
+            // We do not want the connection to fail if the user do
+            // read the property, so if a security exception occurs
+            // continue with the connection.
+            return null;
+        }

         }
     });
 }
```

Fonte: elaborada pelo autor.

Nossas observações mostram que os exemplos são bem variados e dependem muito do contexto em que se inserem. Mostrando exemplos que seriam previsíveis usando métricas na granularidade de arquivo, já outros usando granularidade a nível de função e já em outros casos ao usar nossas métricas de software, poderia trazer resultados inúteis (exemplo 6). O que esse estudo nos mostra é que a previsão de vulnerabilidades é mais complexa do que parece, sendo que vários fatores devem ser levados em consideração, valores das métricas, nível de granularidade e tipo de vulnerabilidade. Por isso é preciso um banco de dados que tenha informação suficiente para realizar explorações separadas por esses fatores. Porém, realizar essas explorações tem custo alto, demandando tempo e trabalho.

Para finalizar, esses exemplos motivacionais nos mostraram alguns *findings*:

1. Uma vulnerabilidade pode não ser prevista ao usar métricas de complexidade de funções, mas pode ter sido escondido ou causada pela complexidade apresentada no arquivo. Talvez, métricas relativas e mixadas entre funções e arquivos poderiam ser implementadas.

2. Alguns tipos de vulnerabilidades apresentam semelhança em código, seguindo certos padrões, e por isso as métricas podem refletir esses padrões, só que adicionar cada tipo de vulnerabilidade a uma amostra pode causar discrepância nas variações dessas métricas escondendo assim esses padrões.

3. Assim como cada tipo de vulnerabilidade apresenta um contexto de código diferente, e verificando esses casos, percebeu-se que nossas métricas de software não representam algumas funções vulneráveis, suspeita-se que sejam certos tipos de vulnerabilidades.

2.4 CONCLUSÃO

Neste capítulo, como referencial teórico, foram conceituados os tipos de vulnerabilidades assim como as métricas que serão estudadas durante o trabalho. Além disso, foi realizado um estudo manual com dez exemplos de vulnerabilidades presentes em funções, relacionando-as a algumas métricas de software que seus valores desviam da normalidade do contexto do projeto. Esse estudo manual nos trouxe várias conclusões sobre o estudo de vulnerabilidades (já citadas na seção anterior), com exemplos de funções que fogem do comportamento padrão que pesquisas realizam e atrapalham modelos preditivos por serem pontos de dispersão.

Contrário a isso, algumas métricas possuíram destaque quando as funções vulneráveis foram destacadas, algumas expressando a complexidade estrutural do código, já outras, deixando implícito critérios de dependência com outros artefatos. Métricas como *CountInput* e *CountOutput* apareceram com certa frequência nas funções vulneráveis, salvo algumas exceções. De certa forma, a quantidade de dados manipulados por artefatos pode sim ser um fator crítico para o desenvolvedor deixar escapar um *Injection* permitindo a exploração de um *Code Execution*.

Outra métrica que se destacou, além da frequência em funções vulneráveis, com valores absurdamente altos, foi *CountPath*. Essa métrica avalia de maneira elementar os caminhos que o código poderá percorrer. De certa forma ela mostra a quantidade de possibilidades de funcionamento de uma função. Segundo a *Reliability Analysis Center* (RAC) (HARTZ; WALKER; MAHAR, 1996), essa métrica também conhecida como *McCabe*'s *cyclomatic complexity* com valor superior a 30 representa uma 'estrutura questionável' e maior que 50 o 'aplicativo não pode ser testado'. A intuição por trás dessa informação é simples, o desenvolvedor precisa pensar nos diversos funcionamentos que o software poderá realizar, funcionamentos inesperados e aleatórios dificilmente trará brechas exteriores, mas a julgar pela displicência e frequência no fechamento de possibilidades, essa proporção poderá aumentar drasticamente as chances dessa brecha aparecer. A quantidade alta do valor dessa métrica, impossibilita o desenvolvedor explorar todas as funcionalidades e comportamento da função ou artefato, podendo esta, se comportar de maneira inesperada e ofensiva, fugindo das políticas de segurança do sistema.

Este capítulo, explanou conceitos, exemplificou vulnerabilidades e motivou ainda mais a pesquisa da relação de como essas métricas de software podem estar correlacionadas a vulnerabilidades em artefatos de código, sendo base para novos conceitos e estudos estatísticos que serão realizados nos capítulos seguintes.

3 CONSTRUÇÃO DO DATASET

Neste capítulo será descrito o processo de coleta de informações. Onde é mostrado os repositórios de vulnerabilidade selecionados, como o processo de *web scrapping*[1] foi realizado, comandos utilizados para tirar *snapshots*[2] dos projetos em repositórios de código, descrição dos projetos analisados, desafios e ferramentas utilizadas para processamento da base de dados.

3.1 PROJETOS ANALISADOS

Foram escolhidos sete projetos *open source*[3], pelo qual foram colhidas métricas e informações sobre eles. Os projetos foram escolhidos por um grau de relevância para comunidade em relação a segurança e consequências críticas para o sistema quando submetidos a ataque. Os projetos são descritos a seguir:

- **Mozilla** é um *web browser* muito utilizado e seu projeto contém módulos menores como o cliente de *e-mail* (Thunderbird), entre outros.
- **Linux Kernel** é o núcleo de um sistema operativo muito utilizado em várias distribuições do Linux (Debian, Fedora, Ubuntu) e utilizado também em outros sistemas.
- **Xen Hypervisor** é um gerenciador de máquinas virtuais utilizado para executar ambientes de virtualização.
- **Httpd** é um servidor de protocolo de comunicação http[4], utilizado em vários sistemas operacionais incluindo UNIX e Windows.
- **Glibc** é uma biblioteca de alta performance escrita em C para executar chamadas de sistemas.
- **Tomcat** é um servidor web desenvolvido para tecnologias da linguagem JAVA.
- **Derby** é um sistema de gerenciamento de banco de dados que pode ser utilizado em programas java, é usado para processamento de transações online.

Os projetos apresentam contextos e tamanhos diferentes, como mostrado na Tabela 2, que mostra a quantidade de linhas de código e quantidade de arquivos de cada projeto. O

[1] *Web scrapping* é uma técnica de engenharia de software utilizada para extrair dados existentes em *websites*.
[2] *Snapshot* de código é como uma fotografia em um determinado momento do código-fonte, congelando assim todas as funcionalidades e implementações do código
[3] Política de código aberto que pode ser adaptado para diferentes fins
[4] Http é um protocolo de comunicação da camada de aplicação para transferência de hipertexto.

Mozilla e o Kernel são os maiores em relação ao tamanho, tendo o Tomcat e o Httpd como os menores. Assim como podem ver, é inviável inspecionar todo o código, devido a quantidade de código em cada projeto, mesmo se o projeto for de pequeno porte como o Httpd.

Tabela 2 – Características dos projetos em estudo.

	Mozilla	**Kernel**	**Xen**	**Httpd**	**Glibc**	**Tomcat**	**Derby**
Linhas	4.145.805	3.068.453	601.330	281.760	1.286.884	265.345	659.358
Arquivos	17.684	13.568	1.558	484	10.531	2.181	2.847
Tempo Coleta	2005-14	2005-15	2012-16	2005-14	2009-15	2005-16	2005-14
Qtde Vuln	870	814	85	48	34	69	53
Domínio	Navegador	SO	Virtualização	Protocolo	Cham Sys	Servidor	SGBD
Linguagem	C/C++	C	C	C	C	Java	Java
Tamanho	Grande	Grande	Pequeno	Pequeno	Médio	Pequeno	Pequeno
Mediana Arq	197	258	192	288	13	117	160
Mediana Func	8	16	14	20	16	4	7

Fonte: elaborada pelo autor.

Observando as linhas referente às medianas, percebe-se que 50% das funções do Kernel estão entre 0 e 16 linhas de código, que apesar de ser um projeto grande, as funções apresentam a mesma frequência que um projeto pequeno como o Glibc que também tem mediana com valor 16. Os outros projetos como Xen, Httpd apresentam medianas 14 e 20, e são projetos relativamente pequenos comparado ao Mozilla com mediana das funções equivalente a 8. Os projetos Tomcat e Derby são escritos na linguagem java e apresentam medianas baixa, respectivamente 4 e 7. Isso mostra uma diversidade nos tamanhos das funções de cada projeto independentemente do tamanho geral do projeto, refletindo mais o perfil de desenvolvimento de cada projeto.

Foi possível colher dados de correções de 1973 vulnerabilidades. Os dados se centram nos dois maiores projetos: Mozilla e Kernel. Isso se reflete um pouco a preocupação com a divulgação e organização para catalogar as vulnerabilidades dos projetos. Ao observar os dados, os intervalos de coleta são bem similares, entretanto a quantidade de vulnerabilidades corrigidas e divulgadas nos repositórios de vulnerabilidade são relativos a cada projeto, dependendo da organização da equipe.

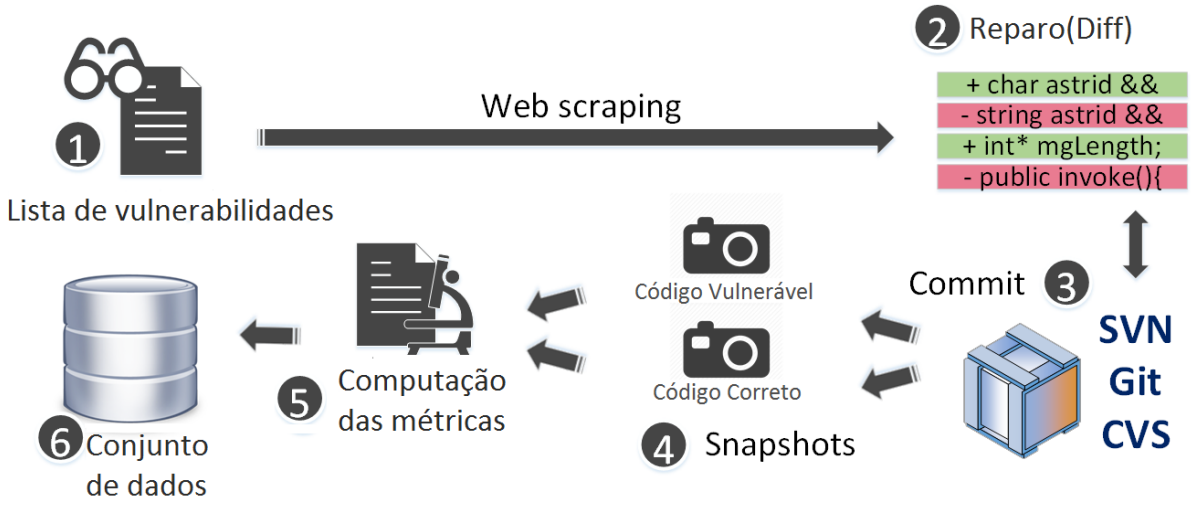
3.2 METODOLOGIA DE COLETA

Para realizar a coleta, foi seguida uma metodologia única para todos os projetos. Em alguns projetos, pulou-se algumas etapas, já outros, tiveram que seguir a risca cada etapa. Mais detalhes serão descritos neste capítulo. Para ilustrar o procedimento, foram colhidas informações do código seguindo a metodologia mostrada na Figura 17:

Os passos dessa metodologia estão descritos a seguir:

- **Passo 1:** Primeiramente, foi verificada a disponibilidade de informações sobre vulne-

Figura 17 – Metodologia de construção da base de dados.



Fonte: elaborada pelo autor.

rabilidades reportadas de cada projeto. Alguns projetos não tinham um repositório de vulnerabilidade ou o mesmo não era de fácil acesso e foram descartados. Porém, os selecionados neste trabalho tem seus dados disponíveis e aptos a serem colhidos. Nesses repositórios tentou-se filtrar o máximo de informação possível, como o tipo de vulnerabilidade que afeta o código e as mudanças necessárias para a correção dessa vulnerabilidade, em forma de *diff*[5]. Esses *commits*[6] foram selecionados porque foram catalogados como *commits* de correção, após observa-los, percebe-se que era pouco provável ter outras mudanças que não fossem para correção da vulnerabilidade. Para chegar a essa conclusão, foram visualizados vários casos manualmente, seus comentários e tamanho da mudança. O processo de *web scrapping* para o *diff* do repositório será explanado na subseção 3.3.

- **Passos 2 e 3:** Esses passos representam as correções necessárias para que aquela vulnerabilidade não afete mais o código. Essa é uma informação importante em todos os repositórios de vulnerabilidade de todos os projetos, pois é a partir dela que é possível direcionar para os *commits* de repositórios de código dos projetos. Essa é uma etapa necessária pois para computar as métricas de software corretamente é preciso todos os artefatos interligados, isso porque algumas métricas são computadas verificando a complexidade estrutural de uma função e das funções chamadas por ela. Os repositórios de vulnerabilidade não nos fornecia essa informação, mostrando apenas as linhas modificadas. De um lado se tem o *diff* destacado no repositório de

[5] diff é um utilitário de comparação de arquivos que gera as diferenças entre dois arquivos, neste caso o arquivo antes da correção da vulnerabilidade e um logo após.

[6] Um commit é o processo de confirmação e divulgação de mudanças experimentais, que no nosso caso é quando um código é atualizado para todos os colaboradores responsáveis.

vulnerabilidade, do outro deve-se encontrar o semelhante em repositório de código, para assim manipular o código como um todo. O processo de identificação dos *commits* no repositório de código foi um grande desafio. Para identificar as funções modificadas para corrigir a vulnerabilidade, foi necessário utilizar o AST[7] na linguagem Java, de forma a identificar as funções que tinham relação com a vulnerabilidade. Além disso, foram utilizados comandos no *git* para varredura de *commits* que apresentavam a mesma mudança no código. Isso porque a assinatura da função não era disponibilizada no repositório de vulnerabilidade, apenas as linhas mudadas.

- **Passo 4:** Identificado o *commit* de correção, o próximo passo é retirar *snapshots* do código. Para nossa pesquisa foram retirados dois *snapshots* em cada momento do *commit* do projeto. Um *snapshot* refere-se ao código antes da vulnerabilidade ser corrigida, ou seja, a vulnerabilidade ainda existia no código. O outro momento era exatamente depois da vulnerabilidade ser corrigida, quando o código não era mais afetado pela vulnerabilidade. Em nossa pesquisa, não foi avaliado o momento depois da vulnerabilidade ser corrigida, mas para trabalhos futuros e para que alterações nas métricas, quando uma vulnerabilidade deixa de existir, possam ser estudadas, essa informação foi armazenada, fazendo com que o banco de dados possa ser expandido conforme o interesse de outras pesquisas.

- **Passo 5:** Com esses *snapshots*, foi possível computar diversas métricas de software a partir de uma ferramenta comercial denominada *Understand*++ (SCITOOLS, 2015). Utilizando-se linhas de comando, esse processo foi automatizado e foram computadas métricas sobre 4.722 *snapshots* de código. Uma descrição das métricas mais utilizadas foi apresentado na Seção 2.2;

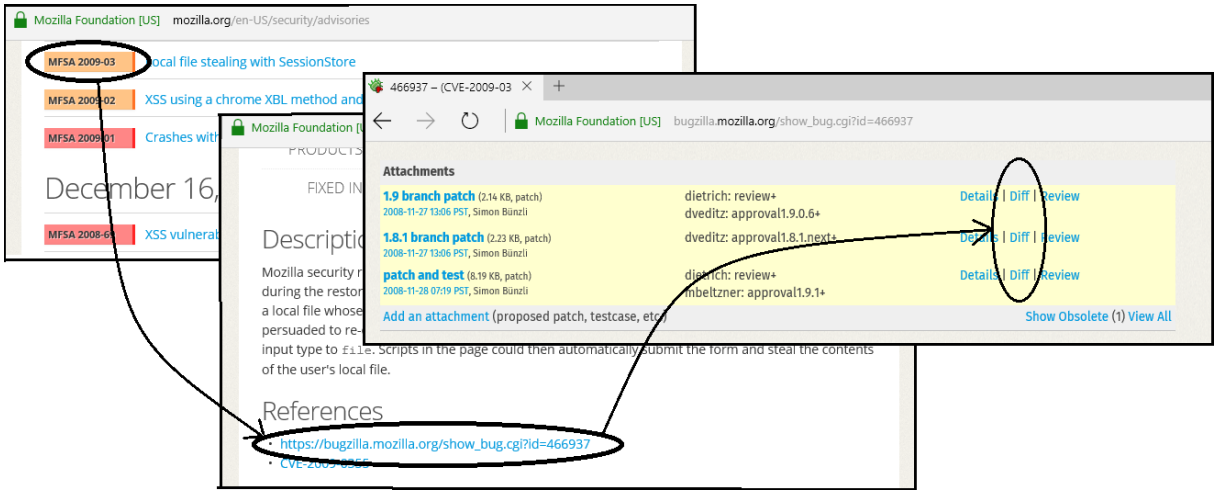
- **Passo 6:** Os dados foram armazenados em um Sistema Gerenciador de Banco de Dados denominado *MySQL* e disponibilizados em (ALVES; ANTUNES; FONSECA, 2016).

3.3 *WEB SCRAPING*

No primeiro momento, tentou-se achar o máximo de vulnerabilidades reportadas em cada projeto. No caso do Mozilla, utilizou-se o site *Mozilla Foundation Security Advisories*[8], navegando por cada registro de vulnerabilidade reportada (MFSA). Cada link desses nos leva a um novo relatório de *bugs* reportados (*Bugzilla*[9]). Em cada *bug* reportado no *bugzilla*, é possível encontrar reparos (*diffs*) que foram necessários para corrigir a vulnerabilidade. A Figura 18 mostra o processo de *web scraping* sobre o repositório de vulnerabilidades do Mozilla.

[7] AST (Abstract Syntax Tree) é a representação da sintaxe do código em formato de uma árvore.
[8] https://www.mozilla.org/en-US/security/advisories/
[9] https://bugzilla.mozilla.org

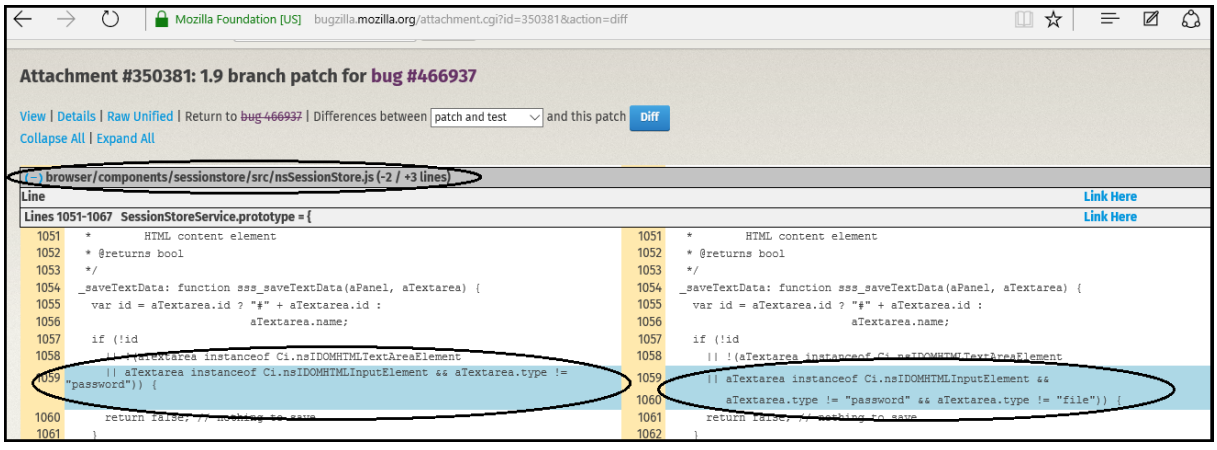
Figura 18 – *Web Scraping* do projeto Mozilla.



Fonte: elaborada pelo autor.

Esses *diffs* mostram as alterações no código, entretanto, não direcionam ao *commit* no repositório do código. Este foi um desafio que foi preciso vencer para realizar essa ligação automática. Como apresentado na Figura 19, foram colhidas linhas adicionadas, removidas ou alteradas e, em seguida, correlacionadas ao respectivo repositório de código referente ao projeto analisado, pois na página não tinham informações que nos levassem diretamente ao *commit* realizado. No caso do Mozilla, é utilizado o repositório *git*[10] pelo qual foi feito clone podendo assim encontrar o *commit* responsável pela correção da vulnerabilidade, que é semelhante ao reportado no site.

Figura 19 – Exemplo de um *Diff*.



Fonte: elaborada pelo autor.

Para realizar esse processo, foi filtrado os *commits* referentes aos arquivos colhidos no site, e verificado um a um qual deles continham aquela mudança, tudo de maneira automática.

[10] https://github.com/mozilla/gecko-dev.git

Identificado o *commit* responsável pela correção, foram retirados *snapshots* do código, que seriam fotografias em determinados momentos do software. Neste caso, foi tirado uma fotografia do código no momento em que a vulnerabilidade foi corrigida, por meio deste comando do *git*:

$$git\ archive - o < caminho > commit$$

Para outro caso, foi tirado um novo *snapshot* do código, mas dessa vez com o código afetado pela vulnerabilidade, utilizando o mesmo comando *git*, adicionando um circunflexo logo após o *commit* responsável pela mudança, assim pegando o código anterior ao da correção. Segue o código a seguir:

$$git\ archive - o < caminho > commit^{\Lambda}$$

Com os *snapshots* do código, a etapa seguinte foi computar métricas de software, tal etapa já foi descrita na Seção 3.2.

Para o Xen Hypervisor, as informações das vulnerabilidades foram coletadas no site Xen Security Advisories[11]. Cada vulnerabilidade reportada está referenciada por ids (XSA e CVE), pelo qual é possível coletar informações. Em cada XSA, é possível navegar entre eles e identificar os reparos (*patches*) responsáveis pela correção da vulnerabilidade. A Figura 20 mostra as páginas navegadas durante o uso da técnica *web scraping*.

Como existe um repositório *git*[12] do Xen, foi colhido o código alterado procurando aloca-los em cada *commit* da mesma forma como foi feito com o Mozilla.

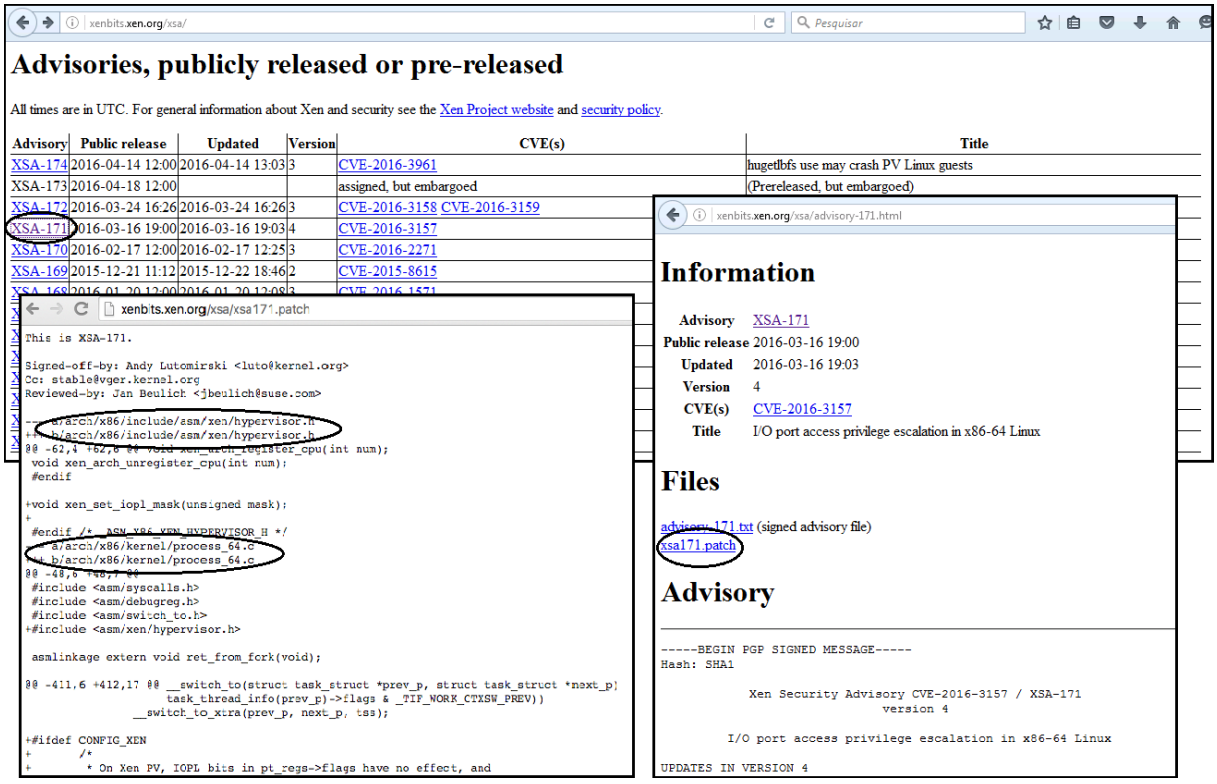
Para o Httpd, Glibc e L.Kernel, foi utilizado uma única abordagem, em que verifica-se todos os CVE Ids referentes a esses projetos, encontrados no site CVE *Details*[13] a fim de encontrar referências à *commits* ou referências ao *bugzilla*. Dessa forma, encontra-se diretamente o *commit* responsável pela correção da vulnerabilidade, assim como mostra a Figura 21.

Para o Tomcat, foram obtidas informações do site da Apache[14], extraindo as revisões e seus respectivos ids no CVE. Com isso tirou-se *snapshots* do repositório de código disponibilizados por eles.

Já o Derby, foram colhidas informação de dados do Artigo de (OHIRA et al., 2015) em que eram apresentados vários tipos de *bugs*. Esses *bugs* foram filtrados para que fossem selecionados os *bugs* de segurança e tirado *snapshots* a partir dos *commits* referentes a eles. Os autores fizeram um trabalho de classificação manual e verificaram todos aqueles *bugs* de

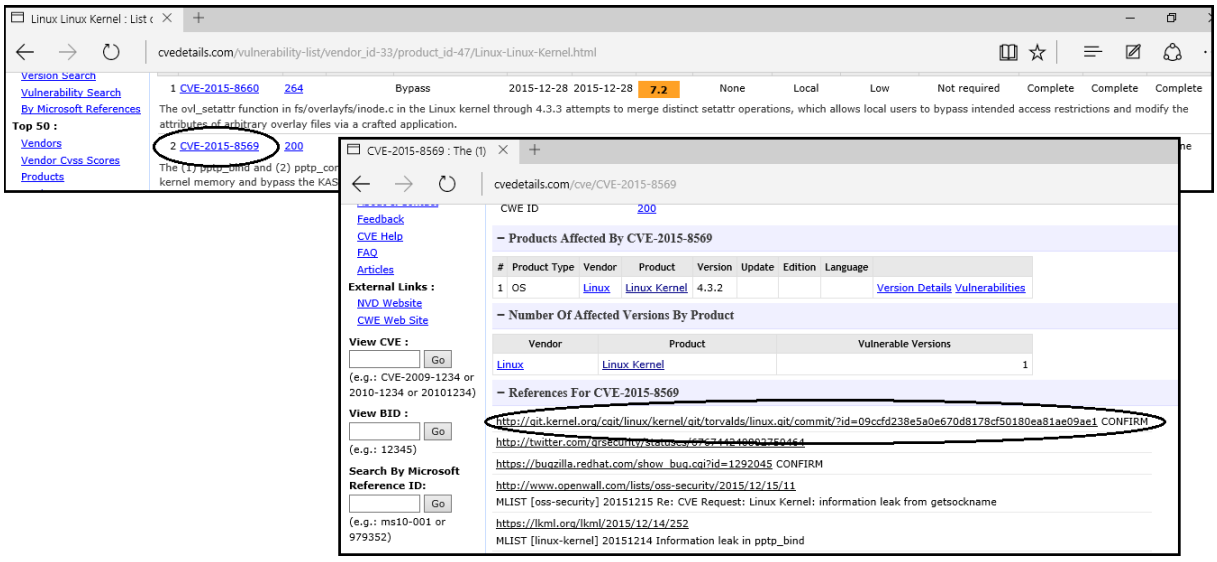
[11] http://xenbits.xen.org/xsa/
[12] git://xenbits.xen.org/xen.git
[13] http://www.cvedetails.com/
[14] https://tomcat.apache.org/security-[6-9].html

Figura 20 – Web Scraping XSA.



Fonte: elaborada pelo autor.

Figura 21 – Web Scraping CVEs.



Fonte: elaborada pelo autor.

segurança, então aproveitou-se parte do trabalho já feito e foi completada com as partes do nosso interesse (*snapshots*, métricas).

3.4 RESUMO DO DATASET

O conjunto de dados usado nas avaliações está presente em (ALVES; FONSECA; ANTUNES, 2016b), e está disponível em um servidor (ALVES; ANTUNES; FONSECA, 2016) online. Esse conjunto de dados contém informações sobre métricas de software referentes a arquivos e funções, e a presença ou não de vulnerabilidade em código. A Tabela 3 apresenta um resumo dos dados contidos nessa base. Foram obtidos informações a partir de 2361 reparos de segurança, correspondendo a 4722 *snapshots* de código.

Tabela 3 – Resumo banco de dados

Projetos	reparos	snapshots	funções		classes		arquivos	
			vul.	total	vul.	total	vul.	total
Mozilla	1123	2246	3.910	7.846.965	1059	1759466	7045	1631788
Kernel	814	1628	7.657	4.168.199	0	526832	19760	1027484
Xen HV	167	334	487	80.173	0	8853	483	20032
Httpd	48	96	117	34.470	0	1918	169	7943
Glibc	34	68	53	50.980	0	4677	177	56983
Tomcat	101	202	1.602	128.495	697	31141	1310	18927
Derby	74	148	1.461	185.360	814	60303	1052	30389
Total	2.361	4.722	15.287	12.494.642	2.570	2.393.190	29.996	2.793.546
# métricas consideradas			28 para c/c++ 22 para java		55 para c/c++ 42 para java		56 para c/c++ 55 para java	

Fonte: elaborada pelo autor.

Alguns pontos importantes devem ser explicados nessa tabela, o primeiro é que existe mais instâncias de arquivos vulneráveis que funções em quase todos os casos. Isso ocorre por dois motivos: o primeiro que algumas vulnerabilidades estão em pedaços de código que não pertencem a função (importação, atributo, configuração), o segundo é que pelo escopo de um arquivo ser maior que de uma função, ele tende a mudar mais que registros de funções e como essa mudança foi gravada como um novo registro, é normal que apareçam mais registros de arquivos vulneráveis que mudanças nas próprias funções. Em resumo, um arquivo que tinha uma vulnerabilidade e mudou 5 vezes os valores de suas métricas serão 5 registros no banco de dados, o mesmo acontece com funções só que elas tendem a mudar em uma frequência menor.

Outro ponto a destacar são os valores nulos para as classes nos projetos Xen HV, Glibc e Kernel. Isso acontece porque são projetos escritos na linguagem predominante C, e a quantidade de classes é extremamente reduzidas e/ou nula. Os valores totais de classes nesses projetos se referem a *structs*, *unions* e outras estruturas com similaridade de classes que o software *understand* cataloga.

Por fim, destacaram-se métricas diferentes dependendo da linguagem de programação abordada. Por exemplo, métricas relacionadas a pré-processamento, só vão existir na

linguagem C/C++. Por esse motivo, a quantidade de métricas colhidas para a linguagem C/C++ foram superiores em cada artefato em comparação com a linguagem java.

4 ESTUDO EXPERIMENTAL

Essa seção está destinada a descrição da realização do estudo experimental baseado na base de dados construída. Serão descritos um estudo exploratório, levantamento de questões de pesquisa, realização de testes estatísticos e exploração de resultados já obtidos. O objetivo deste capítulo é mostrar evidências estatísticas de que há relação entre métricas de software e as vulnerabilidades estudadas.

4.1 QUESTÕES DE PESQUISA

Essa seção está destinada a destacar três questões de pesquisas que serão discutidas nas seções seguintes, para isso usou-se o conjunto de dados criado por nós e projetado um estudo experimental para tentar responder cada uma dessas questões de pesquisa.

- **RQ1:** *As métricas de software tendem a correlacionar-se trazendo informações redundantes?*

  Esse questionamento se dá pelo fato de que, teoricamente, com o aumento do tamanho da função, haverá um aumento dos valores de todas as outras métricas, desta maneira pode estar calculando a mesma informação só que com uso de métricas diferentes. Isso porque, da mesma forma que aumenta-se o tamanho da função, maior ficam as possibilidades e espaços para implementar novas instruções e comandos que podem estar relacionados. Por exemplo, analisando as métricas de contagem de instruções e métricas relacionadas a complexidade ciclomática, o aumento de uma implica no aumento da outra. Para avaliar esses questionamentos, procurou-se por exemplos que apresentam valores distintos e não correlacionados em relação aos valores das métricas. Cada métrica foi correlacionada ao ponto de ver o efeito de correlação que cada métrica exerce sobre as outras. Desta forma é possível observar pontos de congruência entre as métricas e as que realmente se diferem trazendo informações novas sobre os dados da função.

- **RQ2:** *As métricas de software tem poder representativo para distinguir estatisticamente funções vulneráveis de funções neutras?*

  Esse é o grande questionamento deste trabalho. Será mesmo que existe alguma relação estatística entre as métricas de software estudadas neste trabalho e as vulnerabilidades encontradas nos projetos *open source*?

  Para responder esse questionamento, o conjunto de dados foi submetido a alguns testes estatísticos correlacionando os resultados. Esse questionamento é importante para

servir de ponto de partida e incentivo para trabalhos futuros seguindo continuidade a investigação nessa área da engenharia de software.

- **RQ3:** *Vulnerabilidades tendem a reincidir em funções que, anteriormente, já foram afetadas?*

  O objetivo de responder a essa pergunta é verificar se as vulnerabilidades tendem a re-ocorrer em determinados locais do código (funções). Visto que as vulnerabilidades estão mais frequentes em funções mais complexas, é interessante verificar se as vulnerabilidades reincidem por esse motivo.

  Para responder a essa pergunta foram separadas todas as vulnerabilidades e contabilizadas quais delas reincidiram em funções que anteriormente já tiveram uma vulnerabilidade, assim é possível observar as proporções e tirar conclusões.

4.2 REDUNDÂNCIA DAS MÉTRICAS DE SOFTWARE

Esta seção está destinada a esclarecer o questionamento lançado na primeira questão de pesquisa (RQ1), com o objetivo de esclarecer o quão distinto as métricas de software estudadas neste trabalho são, para trazer informações sobre funções em código. O primeiro ponto em destaque é verificar na base de dados se apresenta exemplos com valores distintos. Claramente, ao observar os exemplos destacados na Seção 2.3, percebe-se que à medida que o tamanho da função cresce, cresce também as outras métricas, mas essas outras métricas não crescem proporcionalmente e nem seguindo alguma linearidade. Um exemplo claro pode ser visto na Figura 2, o exemplo 5 e o exemplo 10 apresentam funções com 23 linhas de código e as métricas de contagem de instruções (*CountStmt*) e *tokens* (*CountSemicolon* e *Cyclomatics*) são distintas. Outro exemplo interessante pode ser visto observando a função *OnRedirectStateChange* no exemplo 2 que apresenta 46 linhas de código e a função *createServer* no exemplo 9, que apresentam 53 linhas de código. Ambas apresentam tamanhos semelhantes, mas a distribuição das outras métricas foram bem distintas. Observando as métricas de contagem de *tokens* e instruções não se nota muita diferença, mas ao focar nas métricas mais robustas, nota-se que as métricas *Countpath*, *RatioCommnettoCode* e *CountInput* foram bem distintas. A primeira observação é que a métrica *CountPath* aparece com valor maior que a mediana em *OnRedirectStateChange* que é a função com menor tamanho, e bem baixa em *createServer* que apresenta mais linhas de código. O mesmo acontece para a métrica RatioCommentToCode, mostrando que 29% do tamanho da função *OnRedirectStateChange* são comentários, o que mostra que poucas linhas de código tem poder significativo para demonstrar alta complexidade estrutural de código. Por fim a métrica *CountInput* apresentou valor maior na função *createServer*.

Para concluirmos a observação, verificou-se que as funções em destaque apresentam contextos diferentes, e que as métricas de software conseguiram, de certo modo, representar

a estrutura de cada função. Enquanto foi representativo que a função *OnRedirectStateChange* apresentam mais processamento e mais robustez (representadas pela métricas *CountPath* e *CountStmt* altas), em *createServer* apresenta mais manipulações de dados exteriores a seu escopo (representada pelo valor alto de *CountInput*), e ambas, apesar de distintas, apresentam características de alta complexidade estrutural de código.

Saindo da observação manual, utilizou-se uma técnica estatística para encontrar relações entre essas métricas. É sabido que algumas métricas tem conceitos muito parecidos e podem apresentar dados redundantes. Para identificar esses casos, utilizou-se um algoritmo de correlação denominado *Spearman's Rank Correlation* (HINKLE; WIERSMA; JURS, 2003) e verificou-se a existência de correlação entre esses agrupamentos de métricas que apresentam conceitos similares. O coeficiente de correlação de *Spearman* (*Spearman's Rank Correlation*) é usado para avaliar a dependência estatística entre duas variáveis (HINKLE; WIERSMA; JURS, 2003), não requer distribuição normal de dados e não é afetado por valores extremos, o que é mais adequado para nossos dados. A fim de destacar os extremos dos resultados, a correlação entre duas variáveis foi considerado forte se for superior a 0.9 (HINKLE; WIERSMA; JURS, 2003).

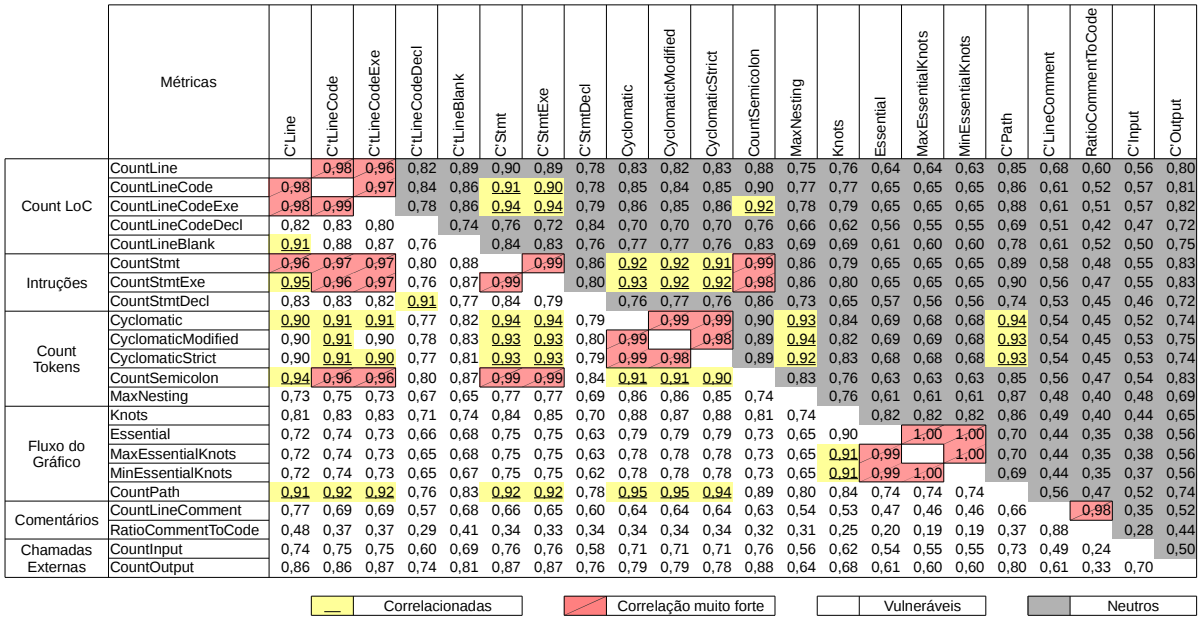
Os resultados dessa análise está descrito na Figura 22, onde é possível visualizar alguns grupos de métricas que apresentam fortes correlações (destacados em amarelo). Nesse estudo, será usado um limiar de 0.95 para o coeficiente de correlação que destacam as métricas que apresentam correlações muito fortes ou quase idênticas, destacando em vermelho as relações pelo qual pelo menos um par[1] supera esse limiar (já que há correlações de funções neutras e funções vulneráveis). Está destacado qualquer valor com coeficiente > 0.9, dividida em dois grupo de coeficientes, representando as instâncias vulneráveis (destacado em branco) e o grupo dos neutros (destacados em cinza), que estão representados na Figura 22.

Se observamos, o tamanho da função, representada por *CountLine*, tem correlação com outras duas tanto para as instâncias neutras como para as vulneráveis, são elas: *CountLineCode* e *CountLineCodeExe*. Isso acontece porque as definições dessas métricas são muito parecidas, praticamente são derivações da métrica *CountLine* eliminando linhas específicas (linhas declarativas ou em branco). Esse resultado confirma nossa intuição tirada pelas observações e responde a primeira questão de pesquisa. A medida que a função cresce, as outras métricas crescem também, porém essas outras métricas tendem a caracterizar a função a partir de informações distintas sobre a função.

Assim como *CountLine* tem suas derivações, *CountStmt* também tem apresentando correlação forte com *CountStmtExe* tanto para o grupo de vulneráveis como para o grupo dos neutros. Outra correlação interessante foi encontrada com uma métrica de contagem de *tokens*, que conta a quantidade de "ponto e virgulas"(*CountSemicolon*), e as métricas de

[1] Um par se refere a correlação entre duas métricas de software, pelo qual um valor é a correlação dessas duas métricas apenas com conjunto de dados vulneráveis, e o outro apenas neutros.

Figura 22 – Correlação entre as métricas



	Métricas	C'Line	C'tLineCode	C'tLineCodeExe	C'tLineCodeDecl	C'tLineBlank	C'Stmt	C'StmtExe	C'StmtDecl	Cyclomatic	CyclomaticModified	CyclomaticStrict	CountSemicolon	MaxNesting	Knots	Essential	MaxEssentialKnots	MinEssentialKnots	C'Path	C'LineComment	RatioCommentToCode	C'Input	C'Output
Count LoC	CountLine		0,98	0,96	0,82	0,89	0,90	0,89	0,78	0,83	0,82	0,83	0,88	0,75	0,76	0,64	0,64	0,63	0,85	0,68	0,60	0,56	0,80
	CountLineCode	0,98		0,97	0,84	0,86	0,91	0,90	0,78	0,85	0,84	0,85	0,90	0,77	0,77	0,65	0,65	0,65	0,86	0,61	0,52	0,57	0,81
	CountLineCodeExe	0,98	0,99		0,78	0,86	0,94	0,94	0,79	0,86	0,85	0,86	0,92	0,78	0,79	0,65	0,65	0,65	0,88	0,61	0,51	0,57	0,82
	CountLineCodeDecl	0,82	0,83	0,80		0,74	0,76	0,72	0,84	0,70	0,70	0,70	0,76	0,66	0,62	0,56	0,55	0,55	0,69	0,51	0,42	0,47	0,72
	CountLineBlank	0,91	0,88	0,87	0,76		0,84	0,83	0,76	0,77	0,77	0,76	0,83	0,69	0,69	0,61	0,60	0,60	0,78	0,61	0,52	0,50	0,75
Intruções	CountStmt	0,96	0,97	0,97	0,80	0,88		0,99	0,86	0,92	0,92	0,91	0,99	0,86	0,79	0,65	0,65	0,65	0,89	0,58	0,48	0,55	0,83
	CountStmtExe	0,95	0,96	0,97	0,76	0,87	0,99		0,80	0,93	0,92	0,92	0,98	0,86	0,80	0,65	0,65	0,65	0,90	0,56	0,47	0,55	0,83
	CountStmtDecl	0,83	0,83	0,82	0,91	0,77	0,84	0,79		0,76	0,77	0,76	0,86	0,73	0,65	0,57	0,56	0,56	0,74	0,53	0,45	0,46	0,72
Count Tokens	Cyclomatic	0,90	0,91	0,91	0,77	0,82	0,94	0,94	0,79		0,99	0,99	0,90	0,93	0,84	0,69	0,68	0,68	0,94	0,54	0,45	0,52	0,74
	CyclomaticModified	0,90	0,91	0,90	0,78	0,83	0,93	0,93	0,80	0,99		0,98	0,89	0,94	0,82	0,69	0,69	0,68	0,93	0,54	0,45	0,53	0,75
	CyclomaticStrict	0,90	0,91	0,90	0,77	0,81	0,93	0,93	0,79	0,99	0,98		0,89	0,92	0,83	0,68	0,68	0,68	0,93	0,54	0,45	0,53	0,74
	CountSemicolon	0,94	0,96	0,96	0,80	0,87	0,99	0,99	0,84	0,91	0,91	0,90		0,83	0,76	0,63	0,63	0,63	0,85	0,56	0,47	0,54	0,83
	MaxNesting	0,73	0,75	0,73	0,67	0,65	0,77	0,77	0,69	0,86	0,86	0,85	0,74		0,76	0,61	0,61	0,61	0,87	0,48	0,40	0,48	0,69
Fluxo do Gráfico	Knots	0,81	0,83	0,83	0,71	0,74	0,84	0,85	0,70	0,88	0,87	0,88	0,81	0,74		0,82	0,82	0,82	0,86	0,49	0,40	0,44	0,65
	Essential	0,72	0,74	0,73	0,66	0,68	0,75	0,75	0,63	0,79	0,79	0,79	0,73	0,65	0,90		1,00	1,00	0,70	0,44	0,35	0,38	0,56
	MaxEssentialKnots	0,72	0,74	0,73	0,65	0,68	0,75	0,75	0,63	0,78	0,78	0,78	0,73	0,65	0,91	0,99		1,00	0,70	0,44	0,35	0,38	0,56
	MinEssentialKnots	0,72	0,74	0,73	0,65	0,67	0,75	0,75	0,62	0,78	0,78	0,78	0,73	0,65	0,91	0,99	1,00		0,69	0,44	0,35	0,37	0,56
	CountPath	0,91	0,92	0,92	0,76	0,83	0,92	0,92	0,78	0,95	0,95	0,94	0,89	0,80	0,84	0,74	0,74	0,74		0,56	0,47	0,52	0,74
Comentários	CountLineComment	0,77	0,69	0,69	0,57	0,68	0,66	0,65	0,60	0,64	0,64	0,64	0,63	0,54	0,53	0,47	0,46	0,46	0,66		0,98	0,35	0,52
	RatioCommentToCode	0,48	0,37	0,37	0,29	0,41	0,34	0,33	0,34	0,34	0,34	0,34	0,32	0,31	0,25	0,20	0,19	0,19	0,37	0,88		0,28	0,44
Chamadas Externas	CountInput	0,74	0,75	0,75	0,60	0,69	0,76	0,76	0,58	0,71	0,71	0,71	0,76	0,56	0,62	0,54	0,55	0,55	0,73	0,49	0,24		0,50
	CountOutput	0,86	0,86	0,87	0,74	0,81	0,87	0,87	0,76	0,79	0,79	0,78	0,88	0,64	0,68	0,61	0,60	0,60	0,80	0,61	0,33	0,70	

Fonte: elaborada pelo autor.

contagem de instruções *CountStmt* e *CountStmtExe*. Isso aconteceu porque as linguagens investigadas tem o caractere ";"como delimitador de instruções, então a correlação mostrou que ambas as métricas, mesmo investigando pontos diferentes, dizem a mesma informação.

Ainda na contagem de *tokens*, foram encontradas correlações entre essas métricas, até pela semelhança entre os tokens de contagem. Por exemplo, nos casos de *Cyclomatic*, *CyclomaticModified* e *CyclomaticStrict*, apresentam correlação muito forte, perto dos 100%, isso porque o que diferem na contagem é a ausência e adição de alguns *tokens*.

Outro grupo de correlações está presente entre as métricas *Essential*, *MinEssentialKnots* e *MaxEssentialKnots* que apresentaram um coeficiente de correlação muito forte, com valor mínimo de 0.99 entre o *Essential* e *MinEssentialKnots* para os vulneráveis e entre *Essential* e *MaxEssentialKnots* com 0.99 também para os vulneráveis, onde o restante dos coeficientes é 1 com máxima correlação.

Por fim, houve correlação entre as métricas *CountLineComment* e *RatioCommentToCode*, mas apenas para o grupo dos neutros, mas que mostra muita importância porque essas métricas falam do mesmo contexto (linhas comentadas) só que uma é relativa e outra é absoluta.

Esses resultados mostram correlações entre determinadas métricas e pode-se pensar em algumas hipóteses que afetam esses resultados. Primeiro que as métricas possuem conceitos muito similares e muitas delas são como métricas derivadas umas das outras, segundo porque outras só mudam a métricas avaliativa (usam média, mínimo, outrora máximo) e são aplicadas ao mesmo contexto. Esses resultados podem influenciar no

desempenho da aprendizagem de máquina. Como solução seria recomendado usar um filtro de correlação para selecionar apenas os atributos mais relevantes desses dados, podendo assim eliminar redundâncias.

4.3 AS MÉTRICAS DE FUNÇÃO E SEU PODER REPRESENTATIVO

Para estudar as diferenças entre os valores das métricas de funções vulneráveis e as funções neutras, foi feita uma análise do nosso conjunto de dados usando o teste estatístico *t-teste*, bem como intervalos de confiança e *box plots*. Em particular, o teste estatístico foi realizado em cada métrica considerada, com o objetivo de entender se os valores do grupo de métricas relacionadas com as funções que contêm vulnerabilidades reportadas foram significativamente diferentes dos valores do grupo de métricas relacionadas com as funções com nenhuma vulnerabilidade reportada. Foi aplicado o teste estatístico considerando duas hipóteses: a hipótese nula que diz que as médias das duas amostras (vulnerável e neutra) são iguais e a hipótese alternativa que elas são diferentes.

A Tabela 4 apresenta os valores dos *p-values* resultantes para cada métrica de função em todos os projetos. O caractere ✓ assinala os casos em que a primeira hipótese tem sido satisfeita, na qual o valor de p é menor do que 0.05. Caso contrário, a célula está marcada com –, então é possível detectar facilmente os padrões.

Tabela 4 – Resultados do *t-test* em todas as métricas, *p-values*.

Métricas	**Mozilla**		**Kernel**		**Xen HV**		**Httpd**		**Glibc**		**Tomcat**		**Derby**	
CountLine	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓
CountLineCode	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓
CountLineCodeExe	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓
CountLineCodeDecl	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓
CountLineBlank	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓
CountStmt	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓
CountStmtExe	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓
CountStmtDecl	→0	✓	→0	✓	0.02	✓	→0	✓	→0	✓	→0	✓	→0	✓
Cyclomatic	→0	✓	→0	✓	→0	✓	→0	✓	0.07	-	→0	✓	→0	✓
CyclomaticModified	→0	✓	→0	✓	→0	✓	→0	✓	0.05	-	→0	✓	→0	✓
CyclomaticStrict	→0	✓	→0	✓	→0	✓	→0	✓	0.08	-	→0	✓	→0	✓
CountSemicolon	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓
MaxNesting	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓
Knots	0.11	-	→0	✓	0.88	-	→0	✓	→0	✓	→0	✓	0.48	-
Essential	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓
MaxEssentialKnots	0.10	-	→0	✓	0.95	-	→0	✓	→0	✓	→0	✓	→0	✓
MinEssentialKnots	0.10	-	→0	✓	0.95	-	→0	✓	→0	✓	→0	✓	→0	✓
CountPath	→0	✓	→0	✓	0.01	✓	0.01	✓	0.02	✓	0.02	✓	→0	✓
CountLineComment	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓
RatioCommentToCode	→0	✓	→0	✓	0.60	-	→0	✓	0.01	✓	→0	✓	→0	✓
CountInput	→0	✓	→0	✓	→0	✓	→0	✓	0.01	✓	0.67	-	→0	✓
CountOutput	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓	→0	✓

Fonte: elaborada pelo autor.

Como se pode observar, os resultados confirmam que há uma diferença significativa entre os valores das métricas na maior parte dos casos. Isso responde, em partes, a nossa questão de pesquisa, mostrando que código com vulnerabilidades tem características diferentes de código sem vulnerabilidades reportadas. É possível ver que, na maioria das métricas a tendência se mantém entre os projetos. Na verdade, apenas em alguns casos (marcados com '-') desviam-se da tendência. Isso é um indício de que as métricas com resultado positivo podem ser capazes de interpretar de forma consistente algumas das características do código que os tornam propensos para conter vulnerabilidades.

O objetivo desse teste é observar se os grupos apresentam semelhança na distribuição normal dos dados (verificando os valores de média amostral), em outras palavras, ele avalia se os grupos, utilizando aquelas característica, são iguais. De fato, o resultado para a maioria dos casos foi positiva, com grande significância que são diferentes, mas em alguns casos como o da métrica *Knots* nos projetos Mozilla, Xen e Derby, não houve significância suficiente para contrariar a hipótese nula, o que não quer dizer que são semelhantes, mas que o teste estatístico não teve força suficiente para comprovar sua diferença.

Além desses, houve casos isolados que se diferenciaram pelo projeto. Foi o caso da métrica *CountInput* para o projeto Tomcat, e *RatioCommentToCode* para o Xen. Isso pode ter acontecido por diversos motivos, podendo ser pela quantidade de instâncias coletadas, ou pela diferença de contextos entre os projetos.

Para visualizar em outra perspectiva, foram observados os *boxplots* e intervalos de confiança de três métricas: *CountLine*, *Cyclomatic* e *Knots*. Duas dessas métricas não tiveram significância em alguns projetos utilizando o teste estatístico para comprovar semelhança entre os grupos de instâncias neutras e o grupo de instâncias vulneráveis. Foi o caso da *Cyclomatic* e suas métricas derivadas para o projeto Glibc e Knots e métricas derivadas para os projetos Mozilla, Xen HV e Derby. As Figuras 23 e 24 mostram os gráficos.

A Figura (23a) apresenta os *boxplots* relativos ao número de linhas (*CountLine*) de funções com e sem vulnerabilidades reportadas ao qual teve resultados positivos no teste estatístico. O padrão é bastante semelhante para todos os projetos: o código neutro apresenta menor mediana e distribuições menos esparsas. Os intervalos de confiança (Figura (24a)) também mostram valores distintos (média) e intervalos independentes. Como padrão, as vulnerabilidades estão presentes em artefatos com maior tamanho, entretanto, não se pode afirmar que essa relação é reciproca, pois o fato da maioria das vulnerabilidades estarem em funções grandes não implica dizer que as funções grandes são responsáveis pela existência da vulnerabilidade.

Já para as métricas que não tiveram resultado positivo em todos os projetos, percebeu-se que os intervalos de confiança e *boxplots* revelam um pouco dessa dispersão dos dados. Isso fica bem claro ao observar a Figura 24b para o projeto Glibc em que aparece intersecção na distribuição dos dados, quando os intervalos de confiança coincidem. Essa dispersão

Figura 23 – Boxplots para código neutro (`N`) e com vulnerabilidades reportadas (`V`).

(a) CountLine (b) Cyclomatic (c) Knots

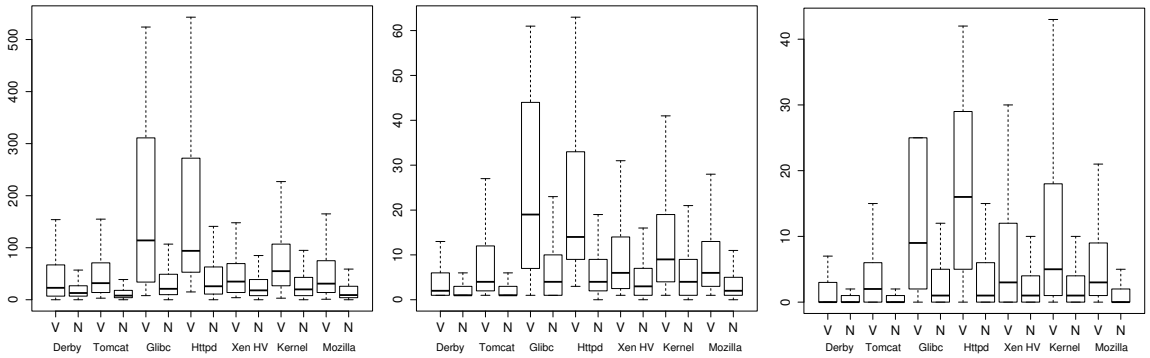
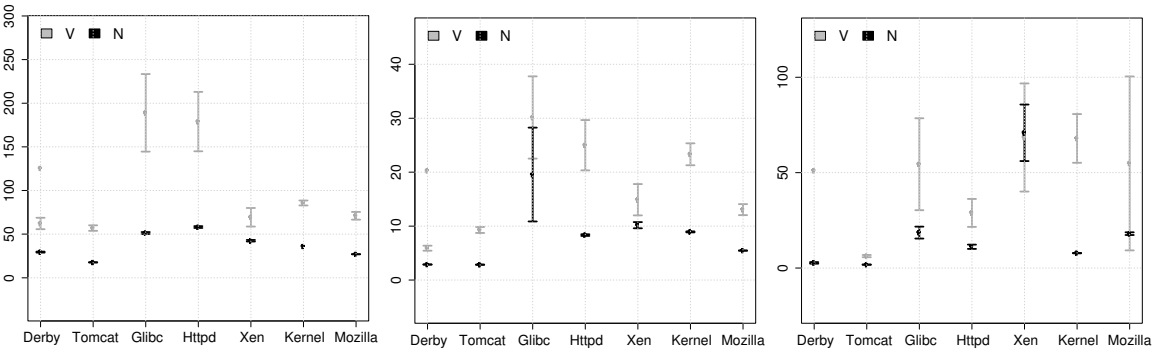


Figura 24 – Intervalos de confiança para código neutro (`N`) e com vulnerabilidades reportadas (`V`).

(a) CountLine (b) Cyclomatic (c) Knots



Fonte: Adaptada de (ALVES; ANTUNES; FONSECA, 2016)

nos dados pode ter comprometido o teste estatístico, que apesar de vermos que a média é distinta, quando aplicado o *t-test*, ele vai a um nível mais profundo padronizando os valores da amostra. Ainda assim a média e mediana para os casos das funções vulneráveis foram superiores as funções neutras, o que é um indicativo positivo para continuarmos investigando essa métrica.

Nossa última métrica neste ponto é a *Knots* que apresentou os piores resultados no teste estatístico *t-test*, resultados ruins nos projetos Xen, Derby e Mozilla. Ao observar seus intervalos de confiança (24c), percebe-se que os dados estão bem dispersos nesses projetos. Os intervalos de confiança se chocam e diferem dos outros tipos de métricas, a média nos casos do Derby e Xen são equivalente e superior respectivamente para as instâncias neutras. O mesmo não ocorre para a mediana que, como podem ver na Figura 23c, as instâncias vulneráveis tem valores superiores em todos os projetos. Isso nos mostra que a métrica *Knots* não segue nenhum padrão aparente e que é preciso mais investigação para verificar se ela discrimina funções vulneráveis, pois falta indícios suficientes que permitam tal afirmação.

Neste sentido, concluímos que há indícios interessantes de que algumas métricas

de software mostram distinção entre funções vulneráveis e neutras. Uma investigação mais profunda dessa linha de pesquisa e vários outros testes e estudos devem ser realizados para aferirmos com veracidade que métricas de software discriminam vulnerabilidades.

4.4 RECORRÊNCIA DE VULNERABILIDADES

O objetivo dessa seção é responder a questão de pesquisa RQ3, que indaga sobre a recorrência de vulnerabilidades em artefatos (funções) de código. Para responder esses questionamentos, foram isoladas todas as vulnerabilidades e as funções que elas afetam e verificado quais delas apareceram em funções que anteriormente já haviam sido afetadas por outra vulnerabilidade.

Ao todo, em nossa base de dados, existem dados sobre 1973 vulnerabilidades reportadas, incluindo todos os projetos. Dessas, 307 vulnerabilidades reincidiram em funções que anteriormente foram afetadas por vulnerabilidades. Visto isso, separou-se todos os casos em cada projeto e foi calculado a proporção de reincidência, essa proporção pode ser melhor vista pela Tabela 5:

Tabela 5 – Proporções de reincidência de vulnerabilidades

Projeto	Recorrência	Vulnerabilidades	Proporção
Mozilla	151	870	17.36%
Kernel	96	814	11.79%
Xen	6	85	7.06%
Httpd	4	48	8.33%
Glibc	1	34	2.94%
Tomcat	37	69	53.62%
Derby	12	53	22.64%
Total	307	1973	15.56%

Fonte: elaborada pelo autor.

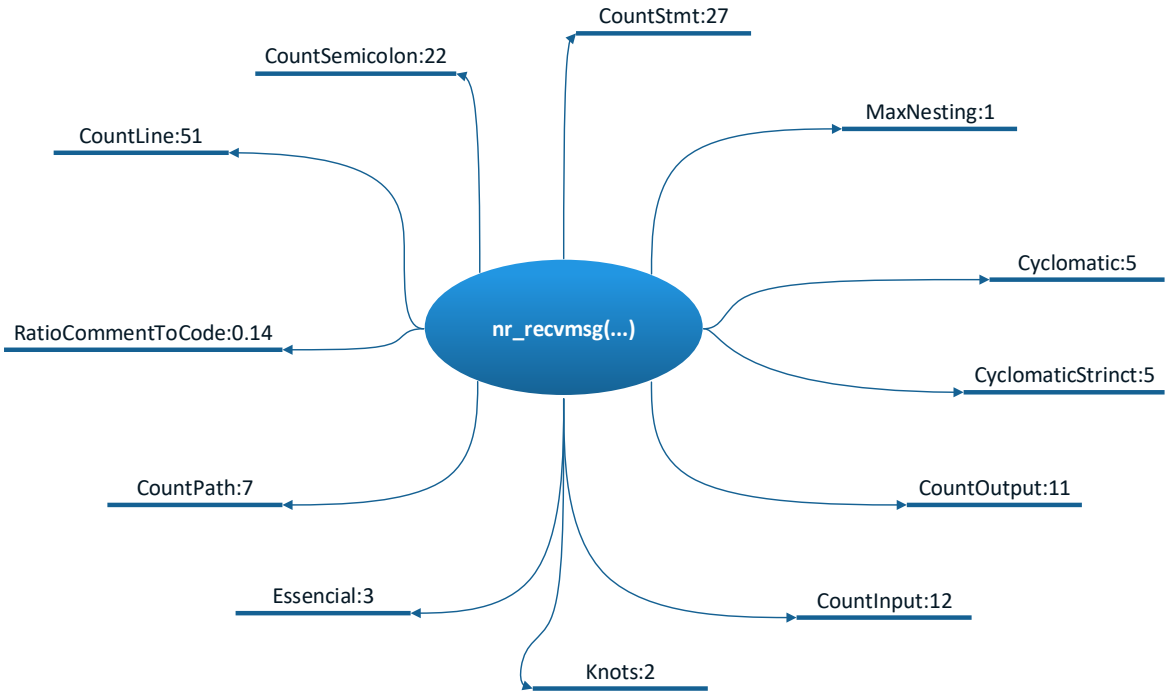
Os projetos escritos em linguagem java foram os que apresentaram maior proporção de reincidência de vulnerabilidades. Isso pode acontecer pela dependência que as funções assumem pela orientação a objetos, ou pela centralização de funcionalidades em certas partes do código, mas ao certo ainda não se pode afirmar isso.

Já os outros projetos mantiveram uma baixa proporção, mesmo quando a quantidade de vulnerabilidades em estudo era considerável, nos casos do Mozilla e do Kernel. No geral, os projetos apresentaram divergência de proporção, o que mostra que nem todos os projetos, essas reincidências ocorrem e suspeita-se que por motivos diferentes, seja pela complexidade estrutural, importância da função ou centralização de funcionalidades.

Para ir um pouco mais além, foi separado a função com maior reincidência de vulnerabilidades para verificar algumas de suas características e métricas. Essa é a função

nr_recvmsg(...), presente no código do Linux Kernel com 8 vulnerabilidades recorrentes e que apresenta os valores de suas métricas com médias mostradas na Figura 25.

Figura 25 – Função com maior reincidência



Fonte: elaborada pelo autor.

Como podem observar, essa não é uma função muito complexa, mas apresenta alguns valores de suas métricas maiores que a média e mediana da amostra do Kernel. As métricas com maior valor para destaque são *CountOutput* que superou a média e mediana da amostra, com valor 11, *CountInput* que superou a mediana com valor 12 mas a mediana da amostra é que é 11 e *RatioCommentToCode* com proporção de 0.14.

CountInput e *CountOutput* são métricas importantes para indicar variáveis de entrada e seus modificadores, dados de entrada podem ser caminhos para chegada de dados maliciosos. Essas métricas determinam a comunicação externa da função e pode mostrar o quão a função está interligada a outras funções ou variáveis. Isoladas, não parecem ter valores altos, mas se interpretadas como funções de chamadas externas e somarmos seus valores seriam um valor razoavelmente alto para o desenvolvedor inspecionar, visto que teria que inspecionar mais 22 funções e/ou variáveis.

Investigando o tipo de vulnerabilidades, nota-se que todas as vulnerabilidades que afetaram essa função foi do tipo *Obtain Information*, essa homogeneidade no tipo de vulnerabilidade é interessante e nos mostra novos caminhos para investigação, porque

reflete a intuição de que certas vulnerabilidades estão mais presentes em funções com contextos específicos. Por exemplo, funções com contexto de exposição ao mundo exterior e são responsáveis pelo contato com a internet estão mais dispostas a sofrerem ataques do tipo *Cross Site Scripting*, ou funções que trabalham com comunicação com banco de dados estão mais predispostas a *Sql Injection* ou ganho de informação.

A fim de concluir a análise e responder a questão de pesquisa RQ3, observou-se que as vulnerabilidades tendem a re-ocorrer mais em projetos escritos na linguagem java, onde o mesmo não ocorre nos escritos em C/C++. Se visto de uma maneira geral, apenas 15% das vulnerabilidades reincidiram em funções que anteriormente já foram afetadas, mas pela diversidade na linguagem e a pouca quantidade de vulnerabilidades em certos projetos, não é possível afirmar isso com veracidade para todo tipo de projeto e linguagem, porque vê-se distinções de resultados como mostrado na Tabela 5. Além disso, não parece que a complexidade da função foi o fator maior responsável pelas reincidência das vulnerabilidades, visto que os valores das métricas de complexidade estrutural do código são baixas.

4.5 CONCLUSÃO

Neste capítulo, foram respondidos três questões de pesquisas que jugou-se relevantes para o estudo deste trabalho. A partir de uma série de estudos estatísticos, avaliou-se a qualidade das *features*, o poder representativo que elas possuem e como as vulnerabilidades tendem a reaparecer em artefatos já anteriormente afetados, a partir de uma série de estudos estatísticos. O ponto principal deste capítulo foi demonstrar que há indícios significantes que mostram que há relação entre métricas de software e vulnerabilidades, mas que essa relação é muito mais complexa do que as associações que muitos trabalhos fazem. A julgar pelos resultados dos testes, essas métricas, após um processo de limpeza e seleção, assim como foi realizado para responder a RQ1, poderão sim servir para discriminar vulnerabilidades, mas não sozinhas. Observando as variações dos resultados em cada projeto, nota-se que essas métricas tendem a se alterar à medida que o contexto do projeto muda, e isso pode nos trazer padrões em certos contextos e anti-padrões em outros, dificultando o estudo centralizado dessas métricas. Nesse caso, *features* que não se prendam tanto a contextos de projetos ou particularidade de desenvolvedores seriam bem vindas.

Como o contexto deste trabalho é o estudo dessas métricas, nos contentamos, por hora, aos resultados positivos que demonstram a relação de significância entre métricas de software e vulnerabilidades em artefatos de código.

5 ANÁLISE COMPARATIVA DOS MODELOS DE PREDIÇÃO

Como um dos objetivos do trabalho, foram replicadas as abordagens de seis estudos envolvendo predição de vulnerabilidades baseadas em métricas de software. Nesta parte de replicação dos trabalhos o interesse está nas métricas de software que representam propriedades estáticas do código. Métricas de execução e métricas de mudança estão fora do nosso escopo. Trabalhos existentes apresentam resultados diversos pois eles usam diferentes configurações e abordagens, considerando até conjuntos distintos de métricas.

Neste contexto, é muito difícil comparar esses trabalhos e selecionar aqueles que são mais adequados para cada cenário de utilização. Além disso, os resultados da avaliação fornecidos em cada uma destas abordagens não são muito útil para avaliar a sua eficácia, por duas razões principais: cada estudo utiliza um conjunto de dados específico para realizar a sua avaliação e seus conjuntos de dados utilizados têm representatividade limitada.

Por estas razões, foram realizadas duas avaliações: (i) examinado as abordagens existentes considerando as configurações e base de dados utilizadas por estas abordagens; e (ii) testando tais abordagens utilizando nossa base de dados. Dessa forma, é possível realizar comparações entre as abordagens uma vez que as mesmas estão sendo analisadas sobre uma única base de dados contendo dados relevantes.

Essa comparação já foi realizada em outro estudo (ALVES; FONSECA; ANTUNES, 2016a), entretanto alguns pontos melhoraram: primeiro que agora ao invés de juntar todos os projetos, foram selecionados apenas um para não ter problemas com o uso de domínios muito distintos, a escolha desse projeto será descrita nas próximas seções. Segundo, que a abordagem agora pode identificar vulnerabilidades existentes em múltiplos *snapshots*, o que quer dizer que funções que antes eram consideradas neutras, agora foram classificadas como vulneráveis porque foi possível estimar a *release* que elas nasceram.

5.1 RESUMO DOS ESTUDOS

As abordagens avaliadas durante nosso estudo vem de trabalhos publicados para construir modelos preditivos usando métricas de software a fim de encontrar as partes de código com a maior probabilidade de ter vulnerabilidades, em todos os trabalhos em questão foi utilizado em nível de granularidade de arquivos. Nesta seção deste trabalho, focou-se em resumir as ferramentas e configurações utilizados pelos estudos que serão comparados em relação a utilização de métricas de complexidade estrutural. Críticas e breve discussão sobre o trabalho será deixado para a seção 8, dos trabalhos relacionados.

Em (SHIN, 2008) é apresentado um estudo de caso usando nove métricas de software. A abordagem submete dados do módulo do Mozilla para treinar um modelo preditivo baseado em *logistic regression*. Seus resultados demonstram que as métricas de complexidade estrutural são capazes de identificar áreas mais propensas a vulnerabilidade, entretanto, tal abordagem ainda precisa ser melhorada, isso porque seus resultados mostraram baixa taxa de falsos positivos e taxa elevada de falsos negativos.

Analogamente, Chowdhury e Zulkernine usam dezessete métricas de software para treinar modelos preditivos, dentre essas, sete são da arquitetura de orientação a objetos (CHOWDHURY; ZULKERNINE, 2011). Os autores utilizaram quatro técnicas de aprendizagem de máquina: *random forest*, *decision trees*, *logistic regression* e *naive bayes*. Utilizaram uma base de dados de vulnerabilidades reportadas no Mozilla em um período de quatro anos. Seus resultados demostram que essas métricas são eficientes para melhorar a predição de vulnerabilidades. Seu modelo preditivo chegou a uma acurácia de 72.85% e recall de 74.22%.

Em (WALDEN; STUCKMAN; SCANDARIATO, 2014) é feito um estudo comparativo entre duas diferentes abordagens, comparando o uso de métricas de software com mineração de texto (*text mining*). Para este caso particular, só foi replicada a abordagem que utiliza as métricas de software. Esse trabalho é interessante porque seus resultados mostraram que *text mining* se sobressaíram aos resultados de métricas de software, entretanto, a comparação dessas abordagens não é tão justa. Existem vários fatores que podem influenciar o desempenho do modelo de predição, como a escolha de métricas de software mais preditivas, os dados de treinamento devem ser bem descritivos, balanceamento dos dados e até mesmo a configuração das técnicas de aprendizagem.

O trabalho (SHIN; WILLIAMS, 2011) compara o uso de métricas de tempo de execução de software com a abordagem de métricas de complexidade estrutural e métricas de dependência de rede. Métricas de dependência de rede são obtidas a partir de um grafo de dependência de rede, onde cada nó representa um arquivo e cada aresta representa uma relação com funções chamadas ou referência entre os arquivos. Nesse estudo, os autores fazem uso de dezenove métricas de software para fazer previsões, entre elas dez são métricas de complexidade, seis são métricas sobre grafo de dependência de rede e três são de execução de código. Os dados de vulnerabilidades foram extraídos de dois projetos, Firefox e Wireshark. O autor usa a técnica de *logistic regression* para comparar as abordagens.

O artigo (SHIN et al., 2011) avalia o desempenho de vinte oito métricas de software para discriminar vulnerabilidades, em que catorze são de complexidade, três são de mudança de código e dez são sobre a atividade do desenvolvedor. Foram usados dados de vulnerabilidade reportados de dois projetos *open source*: Mozilla Firefox e Linux Redhat. Os autores utilizam cinco técnicas de aprendizagem de máquina: *Bayesian Network*, *Decision Tree*, *Naive-Bayes*, *Random forest* e *Logistic Regression*. Entretanto, de acordo com os autores, os resultados são similares àqueles obtidos usando apenas *Logistc Regression* com 3% de

precisão em ambos os projetos, 90% de poder de predição para o Linux e 79% para o Mozilla. Em seus resultados, os autores destacam a elevada taxa de falsos positivos.

Finalmente, em (SHIN; WILLIAMS, 2013) são apresentados três experimentos baseados em vulnerabilidades reportadas no Mozilla. Os autores avaliaram o desempenho de predição de vulnerabilidade a partir de modelos de predição de falhas. Em seus experimentos, os autores treinaram modelos preditivos baseados em *Bayesian Network*, *Decision Tree*, *Logistic Regression* e *Random forest*, mas também afirmaram que seus resultados foram similares ao aplicando apenas *Logistic Regression*.

5.2 CONFIGURAÇÕES OBSERVADAS

Para replicação das abordagens, foram avaliados vários critérios de configuração presentes nos trabalhos destacados, tentou-se replicar, da melhor forma possível, cada abordagem com as informações que foi possível minerar de cada investigação. Essas configurações são descritas a seguir:

- **Métricas de software:** Como já citado, cada trabalho utiliza de conjuntos de métricas de software diferentes. Por exemplo, enquanto nove métricas de software são usadas para construir o modelo preditivo em (SHIN; WILLIAMS, 2008a), dezessete métricas de software foram usadas em (CHOWDHURY; ZULKERNINE, 2011). Então separou-se cada conjunto de dados de acordo com as métricas que cada artigo destaca.

- **Balanceamento:** Para esses tipos de trabalhos envolvendo vulnerabilidade de software, é de se esperar que eles se deparem com o problema de balanceamento dos dados em que o número de instâncias vulneráveis é muito inferior ao número de instâncias neutras. Então a maioria deles tiveram que realizar esse tipo de processo. Apenas em (SHIN, 2008) não foram encontradas informação sobre balanceamento, mas os outros cinco (SHIN; WILLIAMS, 2011; SHIN; WILLIAMS, 2013; SHIN et al., 2011; WALDEN; STUCKMAN; SCANDARIATO, 2014; CHOWDHURY; ZULKERNINE, 2011) utilizaram uma técnica chamada **Random Undersampling (RU)**, que consiste em remover aleatoriamente instâncias da classe majoritária. Outro ponto é que esse processo é aplicado até que a proporção das classes se tornem iguais.

- **Cross validação:** Diferentes abordagens usam diferentes estratégias para a *cross* validação dos resultados. Dentro de alguns casos foi utilizado validação cruzada 3 vezes (**3-fold**) (WALDEN; STUCKMAN; SCANDARIATO, 2014), outro 10 vezes (**10-fold**) (CHOWDHURY; ZULKERNINE, 2011), enquanto em outros casos, esta validação foi repetida 10 vezes, denominada 10x10 *cross* validação (**10x(10-fold)**) (SHIN; WILLIAMS, 2011; SHIN; WILLIAMS, 2013; SHIN et al., 2011). Finalmente, alguns casos usam *next release validation* (**release-fold**) em que o modelo é treinado com versões anteriores e

testado com o próximas versões. Para o caso do (SHIN et al., 2011) a estratégia usada para o Mozilla foi treinar com três versões anteriores e testar na versão corrente, já para o (SHIN, 2008), usava apenas uma versão anterior para testar os dados da versão corrente.

- **Seleção de atributos:** Outro critério de configuração foi a utilização de algoritmos de seleção de atributos. Três trabalhos utilizaram tal abordagem, em que todos os três utilizaram *InfoGainInformation*[1] na qual (SHIN; WILLIAMS, 2011; SHIN et al., 2011) utilizam os 3 melhores resultados e (SHIN; WILLIAMS, 2013) utiliza 7 melhores. Além desses, em (SHIN et al., 2011) acrescentou um algoritmo de seleção baseado em correlação antes de usar o *InfoGainInformation*.

- **Algoritmos de aprendizagem de máquina:** As abordagens utilizam diversos algoritmos de aprendizagem de máquinas utilizados para construir os modelos preditivos. Percebe-se que as mais frequentes são *naive bayes*, *decision trees*, *random forest* e *logistic regression*.

5.3 ANÁLISE DOS TRABALHOS NA LITERATURA

A Tabela 6 resume as abordagens selecionadas, as respectivas configurações e os resultados descritos em cada trabalho analisado. A coluna *Trabalho* mostra as referências dos trabalhos em estudo. A coluna *Amostra* refere-se aos projetos na qual foram coletadas informações sobre os artefatos de código. Já a coluna *Configurações* mostra um resumo das principais configurações utilizadas por cada estudo para construir o modelo preditivo, o restante das colunas são as métricas avaliativas presentes em cada estudo.

Como observados, os resultados são um pouco diversificados, apresentando precisão entre 3.0% e 52.0%, e um recall variando de 29.18% para 90.0%. É interessante notar também que as precisões são baixas e as taxas de falsos positivos são altas, um problema para todas as abordagens avaliadas.

Além disso, todos esses trabalhos foram avaliados utilizando diferentes configurações e a discrepância nos resultados obtidos destaca a necessidade de ter um conjunto de dados comum e representativo para avaliar todos eles. Para isso, utilizou-se a base de dados gerada para realizar um experimento com todos esses projetos e visualizar os novos resultados obtidos.

Essa fase de experimentação consiste em seguir os passos de cada abordagem. Da mesma forma como eles utilizaram, foi utilizado o `weka 3.7.0` para o desempenho do experimento. O weka implementa diversos algoritmos de aprendizagem de máquina em diversas configurações. Cada abordagem utiliza um modelo de classificação binário, ao

[1] Algoritmo de seleção de atributo baseado em entropia.

Tabela 6 – Resumo das configurações encontradas

Abordagens Avaliadas					Resultados Encontrados			
#	**Trabalho**	**Amostra**	**Algoritmo**	**Configurações**	**Acurácia**	**Precisão**	**Recall**	**Taxa de FP**
1	Y. Shin and L. Williams et al (SHIN; WILLIAMS, 2011)	Mozilla	Log. Reg. (T=0.50)	10x(10-fold) ; RU(1:1) ; InfoGain+Ranked(3)	-	9%	71%	-
1	Y. Shin and L. Williams et al (SHIN; WILLIAMS, 2011)	Wireshark	Log. Reg. (T=0.50)	10x(10-fold) ; RU(1:1) ; InfoGain+Ranked(3)	-	11%	78%	-
2	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	Mozilla	Bayesian Network	10x(10-fold) ; RU(1:1) ; InfoGain+Ranked(7)	Similar aos resultados de log reg			
3	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	Mozilla	Decision Tree	10x(10-fold) ; RU(1:1) ; InfoGain+Ranked(7)	Similar aos resultados de log reg			
4	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	Mozilla	Random forest	10x(10-fold) ; RU(1:1) ; InfoGain+Ranked(7)	Similar aos resultados de log reg			
5	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	Mozilla	Log. Reg. (T=0.50)	10x(10-fold) ; RU(1:1) ; InfoGain+Ranked(7)	-	12%	83%	21%
6	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	Mozilla	Log. Reg. (T=0.74)	10x(10-fold) ; RU(1:1) ; InfoGain+Ranked(7)	-	16%	72%	13%
7	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	Mozilla	Log. Reg. (T=0.79)	10x(10-fold) ; RU(1:1) ; InfoGain+Ranked(7)	-	20%	62%	9%
8	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	Mozilla	Log. Reg. (T=0.91)	10x(10-fold) ; RU(1:1) ; InfoGain+Ranked(7)	-	52%	15%	0.6%
9	Y. Shin, A. Meneely (SHIN et al., 2011)	Mozilla	Bayesian Network	Release-fold ; RU(1:1) ; Cor+InfoGain+Ranked(3)	Similar aos resultados de log reg			
10	Y. Shin, A. Meneely (SHIN et al., 2011)	Mozilla	Decision Tree	Release-fold ; RU(1:1) ; Cor+InfoGain+Ranked(3)	Similar aos resultados de log reg			
11	Y. Shin, A. Meneely (SHIN et al., 2011)	Mozilla	Naive-Bayes	Release-fold ; RU(1:1) ; Cor+InfoGain+Ranked(3)	Alto Recall			
12	Y. Shin, A. Meneely (SHIN et al., 2011)	Mozilla	Random forest	Release-fold ; RU(1:1) ; Cor+InfoGain+Ranked(3)	Similar aos resultados de log reg			
13	Y. Shin, A. Meneely (SHIN et al., 2011)	Mozilla	Log. Reg. (T=0.50)	Release-fold ; RU(1:1) ; Cor+InfoGain+Ranked(3)	-	3%	79%	25%
14	Y. Shin, A. Meneely (SHIN et al., 2011)	Kernel Red Hat	Bayesian Network	10x(10-fold) ; RU(1:1) ; Cor+InfoGain+Ranked(3)	Similar aos resultados de log reg			
15	Y. Shin, A. Meneely (SHIN et al., 2011)	Kernel Red Hat	Decision Tree	10x(10-fold) ; RU(1:1) ; Cor+InfoGain+Ranked(3)	Similar aos resultados de log reg			
16	Y. Shin, A. Meneely (SHIN et al., 2011)	Kernel Red Hat	Naive-Bayes	10x(10-fold) ; RU(1:1) ; Cor+InfoGain+Ranked(3)	Higher PD with higher FI			
17	Y. Shin, A. Meneely (SHIN et al., 2011)	Kernel Red Hat	Random forest	10x(10-fold) ; RU(1:1) ; Cor+InfoGain+Ranked(3)	Similar aos resultados de log reg			
18	Y. Shin, A. Meneely (SHIN et al., 2011)	Kernel Red Hat	Log. Reg. (T=0.50)	10x(10-fold) ; RU(1:1) ; Cor+InfoGain+Ranked(3)	-	3%	90%	43%
19	Y. Shin (SHIN, 2008)	Mozilla JS engine	Bayesian Network	Release-fold	Sem informação sobre			
20	Y. Shin (SHIN, 2008)	Mozilla JS engine	Decision Tree	Release-fold	Sem informação sobre			
21	Y. Shin (SHIN, 2008)	Mozilla JS engine	Logistic Reg. (T=0.50)	Release-fold	-	-	12.00%	0.9%
22	Y. Shin (SHIN, 2008)	Mozilla JS engine	Naive-Bayes	Release-fold	Sem informação sobre			
23	J. Walden (WALDEN; STUCKMAN; SCANDARIATO, 2014)	Drupal	Random forest	3-fold ; RU(1:1)	71.0%	52.0%	76.9%	31.6%
23	J. Walden (WALDEN; STUCKMAN; SCANDARIATO, 2014)	PHPMyAdmin	Random forest	3-fold ; RU(1:1)	60.7%	13.2%	66.3%	39.8%
23	J. Walden (WALDEN; STUCKMAN; SCANDARIATO, 2014)	Moodle	Random forest	3-fold ; RU(1:1)	68.3%	1.8%	70.4%	31.7%
24	I. Chowdhury et al (CHOWDHURY; ZULKERNINE, 2011)	Mozilla	Decision Tree	10-fold ; RU(1:1)	72.85%	-	74.22%	28.51%
25	I. Chowdhury et al (CHOWDHURY; ZULKERNINE, 2011)	Mozilla	Logistic Reg. (T=0.50)	10-fold ; RU(1:1)	71.91%	-	59.39%	15.58%
26	I. Chowdhury et al (CHOWDHURY; ZULKERNINE, 2011)	Mozilla	Naive-Bayes	10-fold ; RU(1:1)	62.40%	-	29.18%	4.39%
27	I. Chowdhury et al (CHOWDHURY; ZULKERNINE, 2011)	Mozilla	Random forest	10-fold ; RU(1:1)	72.95%	-	69.43%	23.53%

Fonte: (ALVES; FONSECA; ANTUNES, 2016a)

qual uma classificação é representada por um arquivo com código vulnerável, e a outra classificação é quando o código é neutro pois não apresenta vulnerabilidades reportadas em seu histórico de vulnerabilidades.

O modelo de classificação binária usado pelas abordagens pode gerar dois possíveis erros: falsos positivos e falsos negativos. Um falso positivo é uma instância neutra que tem sido classificadas como vulnerável e este é o código que será inspecionado mas não há nenhuma ameaça de vulnerabilidade. Um modelo com muitos falsos positivo é um modelo que retornar um grande volume de código para inspeção e teste que não deveriam ser focados. Um falso negativo é uma instância vulnerável que foi classificado como neutra, ter falso negativo pode afetar drasticamente a integridade do modelo, assim esse erro deve ser evitado ao máximo, porque representa códigos vulneráveis que não estão mais sendo focados pelo modelo.

5.4 MÉTRICAS PARA AVALIAÇÃO

Durante a análise, foi usado o conjunto de métricas apresentados abaixo para caracterizar a eficácia das abordagens. Estas métricas são calculadas com base nas medições básicas obtidas durante os experimentos: *true positives* (**TP**), *false positives* (**FP**), *true negatives* (**TN**) e *false negatives* (**FN**).

Acurácia (ver expressão 5.1) representa a proporção de casos com classificação correta entre todos os casos. *Taxa de falsos positivos (fpr)* (ver expressão 5.2) representa a taxa de arquivos classificados incorretamente como vulneráveis, dentre todos os arquivos neutros reais. Altas taxas de *fpr* mostra que o modelo retorna muitas instâncias para o desenvolvedor como alarmes falsos.

$$accuracy = \frac{TP + TN}{TP + FN + TN + FP} \tag{5.1}$$

$$fpr = \frac{FP}{FP + TN} \tag{5.2}$$

Para avaliação, foram usadas métricas que têm sido muito utilizadas para caracterizar a eficácia dos sistemas, particularmente no domínio de recuperação de informação. *Precisão* (5.3) representa a taxa de predições corretas dos vulneráveis para todos os arquivos classificados como vulneráveis. *Recall* (5.4) retrata o percentual de arquivos vulneráveis reais dentre todos que foram classificados como vulneráveis. Também chamado de taxa true positivos, ela está relacionada com a forma como o modelo pode cobrir as instâncias vulneráveis. Finalmente, *FMeasure* (5.5) também chamada F_1Score, que representa a média harmônica entre *precisão* e *recall*.

$$precision = \frac{TP}{TP + FP} \tag{5.3}$$

$$recall = \frac{TP}{TP + FN} = tpr \tag{5.4}$$

$$FMeasure = 2 * \frac{precision * recall}{precision + recall} = F_1 Score \tag{5.5}$$

Embora essas métricas resolvam problemas diferentes, todas elas sofrem de algum tipo de viés que pode afetar a sua utilidade e representatividade (ANTUNES; VIEIRA, 2015): 1) **prevalência** de casos positivos; 2) **viés** do modelo; 3) **distorção**; e 4) **relação custo**. Estas distorções nos levou a incluir métricas diferentes em nossa análise.

Inspirado nas probabilidades de apostas (POWERS, 2011), métricas imparciais foram propostas para caracterizar a eficácia dos preditores considerando a proporção das classificações, da mesma forma que é feito com apostas, considerando as duas perspectivas de possibilidades, observando as medidas de vitória (precisão e recall para ganhar) tanto quanto as medidas de perdas (precisão e recall para perder, inverso).

Bookmaker Informedness (5.6) caracteriza a eficácia do preditor considerando medidas da proporção de resultados. As medias de cobertura (recall) tanto para classificar vulnerabilidades quanto recall inverso (detectar neutros) são levados em consideração (5.7). Em outra linha, *Markedness* (5.8) quantifica como consistentemente o resultado que um preditor tem como marcador. Ele utiliza da precisão para detectar vulnerabilidades tanto quanto a precisão para detectar os neutros (5.9).

$$Informedness = \frac{TP}{TP+FN} + \frac{TN}{TN+FP} - 1 \tag{5.6}$$

$$Informedness = Recall + Recall\ Inverso - 1 \tag{5.7}$$

$$Markedness = \frac{TP}{TP+FP} + \frac{TN}{FN+TN} - 1 \tag{5.8}$$

$$Markedness = Precisão + Precisão\ Inversa - 1 \tag{5.9}$$

5.5 EXPERIMENTO COMPARATIVO

Para replicar as abordagens desses estudos, foram escolhidos os dados do Tomcat para submeter a todas as abordagens de predição. Foi preciso escolher um projeto porque aplicar as técnicas em todos os projetos deixa a pesquisa com muita diversidade de instâncias, onde cada projeto apresenta contextos diferentes e podem ter padrões de valores de métricas muito distintos assim como seus limiares, prejudicando assim a eficiência do modelo preditivo.

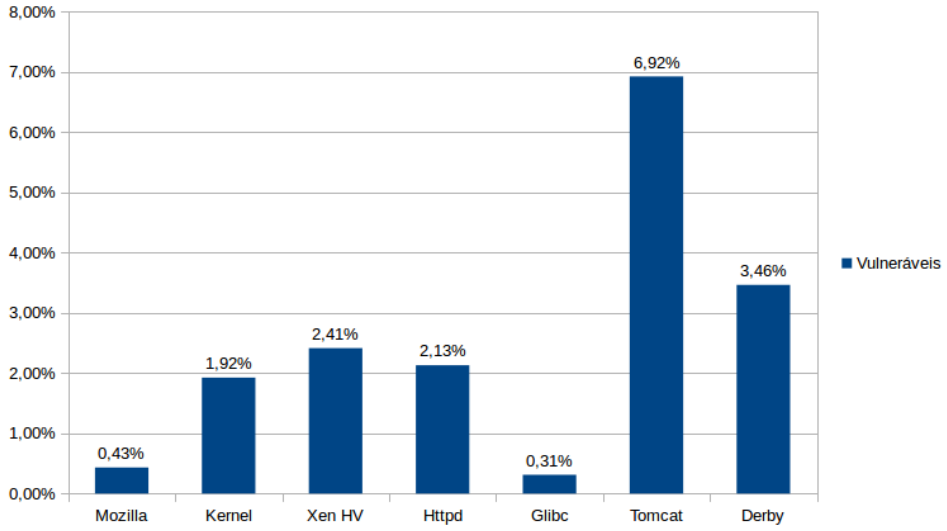
Os dados do Tomcat se sobressaíram entre os demais (Mozilla, Kernel, Xen, Glibc, Httpd, Derby) por um fator: os resultados sofrem fortes distorções com os dados desbalanceados e essa distorção é gradativa, por isso investigou-se a integridade dos dados relacionados a esse problema. A Figura 26 mostra a proporção de registros vulneráveis em relação ao total. Se observar, os dados com menos efeito de desbalanceamento são os dados do Tomcat com proporção de 6.92% de registros vulneráveis. Dessa forma, aplicar os algoritmos aos dados no Tomcat, vão trazer resultados mais positivos, da mesma forma que se aplicado nos projetos que são mais afetados pelo desbalanceamento, se não tratado, trarão resultados insatisfatórios. Por esse motivo o Tomcat foi escolhido para aplicar as abordagens dos trabalhos selecionados.

Aplicando as metodologias dos trabalhos no conjunto de dados do Tomcat, foram obtidos os resultados mostrados na Tabela 7 utilizando cada configuração observada em seus respectivos trabalhos e foram replicados utilizando nossa base de dados referente ao projeto Tomcat.

Os dados foram bem distintos dependendo das configurações observadas. Os valores de precisão foram relativamente baixos, com amplitude de resultados variando de 1.15% à 37.13%. O mesmo é possível falar em relação ao recall, que apesar de seguir uma média de 64.55%, varia de 15.92% à 92.75%.

Os resultados para precisão não foram bons relacionados a outros trabalhos na literatura. Mas isso se da ao fato de que essa métrica sofre grande distorção quando submetida

Figura 26 – Proporção de instâncias de arquivos vulneráveis.



Fonte: elaborada pelo autor.

a proporção desbalanceada dos dados, pois apesar de escolhermos o projeto que apresentava os dados menos desbalanceados, essa proporção ainda fica de 6.92% de classes positivas sobre a quantidade total da amostra. O mesmo pode ser viso nos resultados das abordagem dos artigos que estudamos, entretanto tal fato acontece em menor escala porque a quantidade de informação são inferiores às coletadas na base de dados gerada por nosso trabalho.

Tabela 7 – Resumo dos resultados usando nosso conjunto de dados

Abordagens estudadas				Resultados obtidos (Nos casos de "*" nenhuma instância foi reportada como negativa, então não foi possível computar.)							
#	**Trabalhos**	**Validação**	**Algoritmos**	**Acurácia**	**Precisão**	**Recall**	**Taxa FP**	**Taxa FN**	**F-Measure**	**Informedness**	**Markedness**
1	Y. Shin and L. Williams et al (SHIN; WILLIAMS, 2011)	10x(10-fold)	**Log. Reg (T=0.50)**	68.18%	11.99%	61.79%	31.38%	38.21%	20.09%	30.41%	8.28%
2	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	10x(10-fold)	**Bayesian Network**	70.53%	13.52%	65.82%	29.14%	34.18%	22.43%	36.67%	10.29%
3	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	10x(10-fold)	**Decision Tree**	75.93%	11.34%	39.86%	21.58%	60.14%	17.65%	18.29%	6.30%
4	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	10x(10-fold)	**Random forest**	87.28%	12.41%	15.92%	7.78%	84.08%	13.95%	8.14%	6.47%
5	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	10x(10-fold)	**Log. Reg (T=0.50)**	57.02%	10.30%	73.16%	44.10%	26.84%	18.06%	29.06%	7.08%
6	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	10x(10-fold)	**Log. Reg (T=0.74)**	26.55%	7.13%	86.02%	77.57%	13.98%	13.17%	8.45%	2.99%
7	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	10x(10-fold)	**Log. Reg (T=0.79)**	21.50%	6.75%	86.84%	83.02%	13.16%	12.53%	3.82%	1.66%
8	Y. Shin and L. Williams (SHIN; WILLIAMS, 2013)	10x(10-fold)	**Log. Reg (T=0.91)**	13.94%	6.36%	89.61%	91.29%	10.39%	11.88%	-1.68%	-1.27%
9	Y. Shin, A. Meneely (SHIN et al., 2011)	Release-fold	**Bayesian Network**	73.82%	4.96%	73.26%	26.17%	26.74%	9.30%	47.09%	4.29%
10	Y. Shin, A. Meneely (SHIN et al., 2011)	Release-fold	**Decision Tree**	83.30%	5.41%	49.27%	16.07%	50.73%	9.75%	33.20%	4.30%
11	Y. Shin, A. Meneely (SHIN et al., 2011)	Release-fold	**Naive-Bayes**	82.98%	6.70%	64.12%	16.67%	35.88%	12.13%	47.46%	5.90%
12	Y. Shin, A. Meneely (SHIN et al., 2011)	Release-fold	**Random forest**	89.76%	7.06%	37.69%	9.26%	62.31%	11.89%	28.43%	5.79%
13	Y. Shin, A. Meneely (SHIN et al., 2011)	Release-fold	**Log. Reg (T=0.50)**	83.53%	6.77%	62.56%	16.08%	37.44%	12.21%	46.48%	5.94%
14	Y. Shin, A. Meneely (SHIN et al., 2011)	10x(10-fold)	**Bayesian Network**	65.45%	12.47%	72.05%	35.01%	27.95%	21.26%	37.04%	9.58%
15	Y. Shin, A. Meneely (SHIN et al., 2011)	10x(10-fold)	**Decision Tree**	67.11%	13.19%	73.07%	33.30%	26.93%	22.34%	39.77%	10.47%
16	Y. Shin, A. Meneely (SHIN et al., 2011)	10x(10-fold)	**Naive-Bayes**	75.73%	13.47%	50.67%	22.53%	49.33%	21.28%	28.14%	9.25%
17	Y. Shin, A. Meneely (SHIN et al., 2011)	10x(10-fold)	**Random forest**	80.05%	17.82%	57.64%	18.40%	42.36%	27.23%	39.24%	14.35%
18	Y. Shin, A. Meneely (SHIN et al., 2011)	10x(10-fold)	**Log. Reg (T=0.50)**	65.29%	11.68%	66.50%	34.80%	33.50%	19.87%	31.71%	8.25%
19	Y. Shin (SHIN, 2008)	Release-fold	**Bayesian Network**	78.36%	7.25%	74.78%	21.56%	25.22%	13.22%	53.22%	6.53%
20	Y. Shin (SHIN, 2008)	Release-fold	**Decision Tree**	84.05%	10.34%	81.24%	15.89%	18.76%	18.34%	65.35%	9.84%
21	Y. Shin (SHIN, 2008)	Release-fold	**Log. Reg (T=0.50)**	80.16%	7.42%	69.71%	19.61%	30.29%	13.41%	50.10%	6.58%
22	Y. Shin (SHIN, 2008)	Release-fold	**Naive-Bayes**	91.56%	10.79%	38.88%	7.25%	61.12%	16.89%	31.63%	9.32%
23	J. Walden (WALDEN; STUCKMAN; SCANDARIATO, 2014)	3-fold	**Random forest**	66.88%	1.15%	73.43%	33.15%	26.57%	2.27%	40.27%	0.94%
24	I. Chowdhury et al (CHOWDHURY; ZULKERNINE, 2011)	10-fold	**Decision Tree**	80.32%	22.66%	84.50%	19.97%	15.50%	35.73%	64.54%	21.33%
25	I. Chowdhury et al (CHOWDHURY; ZULKERNINE, 2011)	10-fold	**Log. Reg (T=0.50)**	74.71%	15.26%	63.82%	24.53%	36.18%	24.63%	39.29%	12.05%
26	I. Chowdhury et al (CHOWDHURY; ZULKERNINE, 2011)	10-fold	**Naive-Bayes**	84.76%	18.00%	38.09%	12.01%	61.91%	24.45%	26.08%	13.36%
27	I. Chowdhury et al (CHOWDHURY; ZULKERNINE, 2011)	10-fold	**Random forest**	89.37%	37.13%	92.75%	10.87%	7.25%	53.03%	81.88%	36.57%

Fonte: elaborada pelo autor.

Para exemplificar, comparando os resultados com aqueles apresentados na Tabela 6, foi percebido que as precisões assumem valores bem distintos, mas isso se dá devido as proporções do conjunto de dados utilizados no trabalho de cada autor. Os melhores resultados foram atingidos pelo trabalho de (WALDEN; STUCKMAN; SCANDARIATO, 2014), que trabalha com projetos pequenos, o número de vulnerabilidade para os projetos colhidos foram 75

vulnerabilidade para o PHPMyAdmin, 51 para o Moodle e 97 para o Drupal, tendo que utilizar estratégia de validação 3-*fold* pela quantidade pequena de instâncias vulneráveis.

Já em (SHIN; WILLIAMS, 2013), uma base de dados com 363 arquivos vulneráveis foi utilizada, em que apenas 3% dos arquivos eram vulneráveis. Percebe-se que o valor da precisão chegou a valores como 52% e 20% (ver Tabela 6), mas ao custo de recall, alterando o valor do limiar de classificação da *logistic regression*. Em (SHIN; WILLIAMS, 2011) são utilizados dois projetos em que o dataset contém 301 arquivos vulneráveis para o Mozilla e 181 arquivos vulneráveis para o Wireshark contemplando 3.8% e 7.8% dos arquivos dos projetos respectivamente e suas precisões não superaram 12%.

O intuito é mostrar que, à medida que se colhe uma instância vulnerável, dependendo do projeto, serão colhidas milhares que não são vulneráveis e isso aumenta gradativamente. O conjunto de dados coletados por este trabalho contém 1973 vulnerabilidades colhidas.

Outro ponto de extrema importância é o balanceamento dos dados, que foi uma escolha unânime para todos os trabalhos utilizando de *ramdom undersampling* para realizar esse balanceamento. Esse tipo de técnica pode causar a perda de informação de grande relevância. Mesmo repetindo o processo 10 vezes, ainda se tem grandes chances dos dados estarem enviesados pela seleção de poucas instâncias em um universo grande de possibilidades. Nossa sugestão é utilizar de filtros mais robustos como uma abordagem estratificada junto com algum algoritmo de agrupamento para caracterizar as instâncias similares, claro que desempenho e custo devem ser levados em consideração.

Em relação ao recall, vê-se resultados bem interessantes, onde a maioria resultou em recall superior à 60%, chegando a casos próximos de 90%. Porém, recall é também uma métrica que é influenciada pelo balanceamento.

Para avaliar melhor os dados com tantas distorções de resultados, utilizou-se duas métricas emprestadas das probabilidades de jogatina, pelo qual se calcula, em nosso contexto, a porcentagem tanto de se detectar uma instancia vulnerável, como a porcentagem de detectar uma instancia neutra, sendo assim métricas melhores para avaliação do nosso caso.

Informedness e *markedness* mostram resultados relativamente interessantes, o *informedness* com valor máximo de 81.88% mostra um alto poder de cobertura para a amostra submetida (Tomcat), considerando tanto a cobertura para detecção de vulneráveis como neutros. Em relação a *Markedness* os valores são menores, com o máximo chegando apenas a 36.57%, isso porque essa métrica está relacionada a precisão de detecção. Mas vale a pena ressalvar que essas métricas variam de -1 a 1 diferentemente de métricas como *precision* e *recall* que apresentam intervalos válidos de 0 a 1. Logo esses valores apesar de parecerem pequenos, se estão positivos já superam a média do intervalo que as métricas adotam.

O melhor resultado foi apresentado nas estatísticas do caso 27. Para todas as métricas

estatísticas, seu resultado foi superior aos demais. Resumindo as configurações, percebe-se que foi utilizado *random forest* e 10-*fold validation*. Outros trabalhos como no caso 4 e 17 também utilizaram essa configuração, entretanto repetindo 10 vezes, o 10-*fold validation*. Essa mudança de repetição, supostamente não foi o ponto chave para tal melhoria. Investigando as métricas utilizadas por essas configurações, é possível ver que os resultados se sobressaíram quando fora utilizadas as seguintes *features*: *AvgCyclomatic*, *SumCyclomaticStrict*, *SumCyclomaticModified*, *SumEssential*, *CountLineCode*, *MaxMaxNesting*, *CountPath*, *FanIn*, *FanOut*, *HK*, *CountDeclFunction*, *DIT*, *NOC*, *CBC*, *RFC*, *CBO* e *LCOM*. Já os outros casos (4 e 17) selecionaram 7 melhores utilizando *InfoGainInformation* e 3 melhores respectivamente, dentre as *features*: *CountLineCode*, *CountDeclFunction*, *CountLineCodeDecl*, *CountLinePreprocessor*, *SumEssential*, *SumCyclomaticStrict*, *SumMaxNesting*, *MaxCyclomaticStrict*, *MaxMaxNesting*, *FanIn*, *FanOut*, *MaxFanIn*, *MaxFanOut* e *RatioCommentToCode*.

5.6 CONCLUSÃO

Este capítulo foi destinado a realização de um estudo comparativo entre os modelos de predição de vulnerabilidades por meio de métricas de software. O objetivo foi mostrar a disparidade entre os resultados de cada modelo e como eles podem ser distintos ao utilizar a mesma base de dados. Além disso, mostrar como as métricas de medições atuais podem não encaixar perfeitamente quando o contexto a ser estudado são vulnerabilidades de software que pela escassez de exemplos, podem trazer problemas específicos como o balanceamento dos dados.

É importante notar que são vários os aspectos que podem ser mudados e configurados para encontrar maneiras de melhoria nesses resultados. Como forma de comparação, foram aplicados modelos engessados e que foram modelados para determinado contexto. Mas ao diminuirmos para granularidade de funções, esses desafios continuarão, ou até mesmo se tornarão piores, pois as proporções ficarão mais extremas.

6 SELEÇÃO DE MÉTRICAS

Neste capítulo será realizado procedimentos para seleção e redução de dimensionalidade da amostra. Assim como já realizado em alguns artigos (STUCKMAN; WALDEN; SCANDARIATO, 2017), serão utilizados duas técnicas distintas já estudadas e de grande relevância na literatura: análise de componentes principais e ganho de informação a partir da importância das *features*.

6.1 ELIMINANDO REDUNDÂNCIA DOS ATRIBUTOS

Como mostra na Seção 4.2, algumas métricas apresentam forte correlação e isso mostra que são dados redundantes no *dataset*. Para submissão desses algoritmos a técnicas estatísticas, foi selecionado apenas uma característica, dentro das que resultaram com altíssima correlação a fim de simplificar a amostra. Para isso foi utilizado um algoritmo que lista a importância de cada atributo e investigado qual dentre eles possui mais influência na descrição do nosso problema.

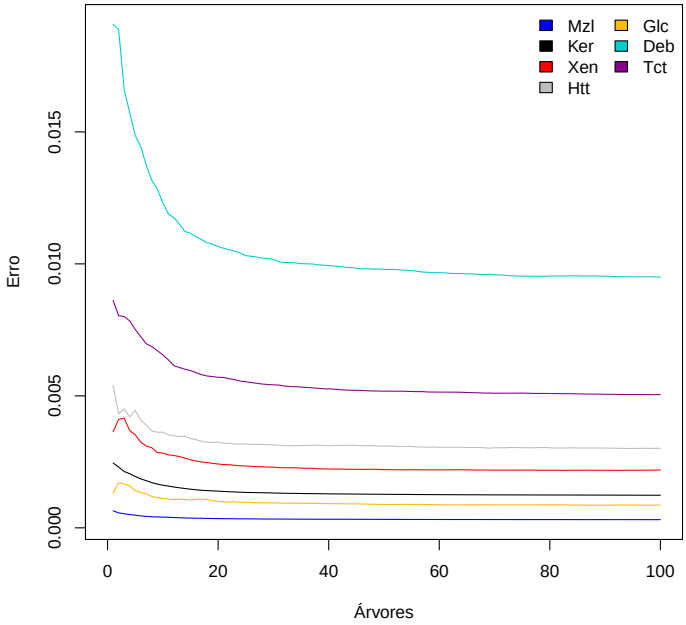
O algoritmo utilizado é o *Random Forest* que mensura uma variável baseada na importância de cada atributo com a aprendizagem da técnica. Esse algoritmo foi escolhido porque é mais robusto a ruídos que outras técnicas que também se baseiam em entropia como Árvore de Decisão (DT), e esses dados apresentam alto grau de ruído (*outliers*, alta variância).

As configurações utilizadas foram: 50 como número de árvores geradas, as variáveis de entrada selecionadas aleatoriamente em cada nó são definidas de acordo com o padrão da função em R que para nosso caso divide o número de atributos num total de 3, resultando em 4; e cada árvore é deixada crescer na maior extensão possível, isto é, a profundidade máxima da árvore é ilimitada. O atributo de maior influência é o número de árvores gerada e foi escolhido a partir de um estudo da diminuição de erro da amostra como mostra a Figura 27.

Se observamos, ao aumentar o número de árvores geradas, não há diferença significativa na diminuição do erro estimado, apenas o Derby ainda houve variação no erro. Assim é possível utilizar de um modelo mais simples com 50 árvores de decisão e assim passar para próxima etapa que é executar o algoritmo nos casos selecionados.

O primeiro grupo a ser analisado é referente as métricas de contagem de linhas, são elas: *CountLine*, *CountLineCode* e *CountLineCodeExe*, que apresentaram correlação muito forte. A Tabela 8 mostra o resultado da aplicação de um algoritmo que calcula grau de importância usando *Random Forest*.

Figura 27 – Numero de árvores em função da taxa de erro.



Fonte: elaborada pelo autor.

Tabela 8 – Importância das métricas de contagem de linhas com alta correlação

Métrica	Mozilla	Kernel	Xen HV	Httpd	Glibc	Tomcat	Derby	Total
C'Line	4,69	**23,89**	**6,67**	2,78	1,99	2,27	**5,30**	**47,59**
C'LineCode	**10,47**	5,58	3,15	1,64	1,70	**4,20**	6,44	33,17
C'LineCodeExe	8,62	5,87	4,83	**3,29**	**2,11**	2,24	3,07	30,03

Fonte: elaborada pelo autor.

Como observado, a métrica *CountLine* se sobressaiu dentre as demais em um contexto geral, entretanto, verificando isoladamente cada projeto, observa-se que a métrica com mais importância varia, apesar desse fato, selecionou-se a métrica *CountLine* porque apresenta maior valor na maioria dos casos e quando junta-se todos os projetos.

O segundo grupo a ser observado são das métricas: *CountSmtm*, *CountStmtExe* e *CountSemicolon*. A Tabela 9 mostra os valores de importância de cada métrica em cada um dos projetos estudados.

Observando os valores, os dados ao final foram bem equilibrados para as métricas *ContStmt* e *CountStmtExe*, entretanto a métrica *CountSemicolon* não teve resultados próximos. *CountStmtExe* se sobressaiu em vários projetos, entretanto seus valores foram bem parecidos com as demais métricas quando isso acontecia e quando se trata de quantidade de vulnerabilidades atingidas. Os projetos Kernel e Mozilla tem maior grau de importância e a métrica que se sobressaiu nesses projetos foi *CountStmt* de uma maneira geral, sendo ela a

Tabela 9 – Importância das métricas de contagem de instruções com alta correlação

Métrica	Mozilla	Kernel	Xen HV	Httpd	Glibc	Tomcat	Derby	Total
C'Stmt	**4,74**	**11,21**	4,26	2,44	2,06	5,28	4,79	**34,77**
C'StmtExe	4,56	7,91	**5,16**	**3,13**	**2,27**	**7,27**	4,06	34,36
C'Semicolon	2,79	6,81	3,59	2,78	2,16	5,48	**5,92**	29,53

Fonte: elaborada pelo autor.

escolhida para a continuidade do trabalho.

O terceiro grupo são das métricas de contagem de *tokens*. As métricas com alta correlação são: *Cyclomatic*, *CyclomaticModified* e *CyclomaticStrict*. A Tabela 10 mostra os resultados referentes a importância de cada métrica.

Tabela 10 – Importância das métricas de contagem de *tokens* com alta correlação

Métricas	Mozilla	Kernel	Xen HV	Httpd	Glibc	Tomcat	Derby	Total
Cyclomatic	3,66	7,45	5,10	2,49	**2,16**	2,37	4,99	28,23
Cyclo'Modified	3,00	10,87	**5,70**	**3,09**	2,01	**5,72**	3,89	34,28
Cyclo'Strict	**11,11**	**13,40**	2,37	1,72	1,85	3,84	**5,60**	**39,87**

Fonte: elaborada pelo autor.

Investigando os resultados, percebeu-se que a métrica *CyclomaticStrict* se sobressaiu as demais. Diferentemente dos resultados anteriores, nesse a métrica, supostamente derivada, obteve mais valor de importância geral. Ao investigarmos sua definição, nota-se que ela conta mais *tokens* que as demais, o que pode ter possibilitado capturar mais descrição do atributo classificador em estudo.

Como quarto grupo de alta correlação destacam-se as métricas: *Essential*, *MinEssentialKnots* e *MaxEssentialKnots*. A Tabela 11 mostra os resultados.

Tabela 11 – Importância das métricas de controle de fluxo com alta correlação

Métricas	Mozilla	Kernel	Xen HV	Httpd	Glibc	Tomcat	Derby	Total
Essential	7,78	3,91	**7,32**	2,55	**2,98**	3,13	4,04	31,72
MaxEssentialKnots	**9,10**	**6,78**	5,79	2,30	1,69	**6,16**	3,86	**35,68**
MinEssentialKnots	5,57	2,40	3,39	**4,60**	1,52	3,49	**6,71**	27,69

Fonte: elaborada pelo autor.

Nesse grupo de métricas, a métrica *MaxEssentialKnots* teve os melhores resultados de importância. Foi superior em 3 projetos e de um modo geral. Nesse caso, essa métrica será mantida representando todas as métricas do grupo.

Por fim, será abordado o grupo das métricas de contagem de linhas comentadas. São apenas duas, mas que apresentaram alta correlação. O valor dos resultados estão apresentados na Tabela 12. Os resultados mostraram que usar a taxa de linhas comentadas se sobressai ao

valor absoluto de linhas comentadas na maioria dos casos e na importância geral. Então, a métrica *RatioCommentToCode* foi selecionada para representar esse grupo de métricas com alta correlação.

Tabela 12 – Importância das métricas de medida de comentários com alta correlação

Métricas	Mozilla	Kernel	Xen HV	Httpd	Glibc	Tomcat	Derby	Total
C'LineComment	3,60	5,83	4,29	1,68	**4,26**	4,77	**8,69**	33,13
RatioC'ToCode	**9,11**	**9,24**	**4,59**	**1,84**	2,39	**6,43**	7,22	**40,83**

Fonte: elaborada pelo autor.

A partir desse estudo, eliminou-se algumas características redundantes e foi simplificado o conjunto de *features* do nosso banco de dados. Em resumo, simplificou-se 14 atributos em 5. Destacaram-se então as métricas *CountLine*, *CountStmt*, *CyclomaticStrict*, *MaxEssentialKnots* e *RatioCommentToCode*. Além dessas, serão feitas investigações juntamente com as que foram independentes no teste de correlação, são elas: *CountLineCodeDecl*, *CountLineBlank*, *CountStmtDecl*, *MaxNesting*, *Knots*, *CountPath*, *CountInput* e *CountOutput*. Nos restando assim ainda 13 métricas para investigação.

6.2 SELEÇÃO DE ATRIBUTOS

Nesta seção são realizados estudos para selecionar as melhores *features* dentre as que permanecerem após a verificação de correlação entre as métricas. Para tal, foram utilizadas duas abordagens para realização dessa atividade, são elas: uma técnica de seleção de atributos por meio da importância, assim como foi feito para simplificar os grupos de métricas com alta correlação e análise de componentes principais (PCA) (ABDI; WILLIAMS, 2010).

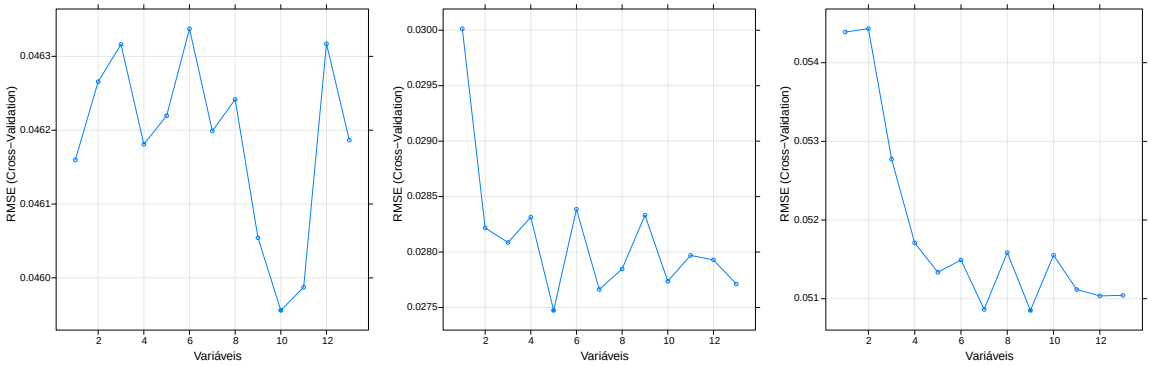
Para realizar a seleção por meio da importância, foi utilizado um método automático de seleção de *features* denominado *Recursive Feature Elimination* (RFE) juntamente com o uso da importância do *Random Forest* para realizar o *rank* das *features* (GRANITTO et al., 2006), utilizando da mesma configuração investigada na Seção 6.1.

O algoritmo retorna as melhores *features* dependendo de certas métricas. Assim, foi escolhida a métrica RMSE (*root-mean-square deviation*) para selecionar as melhores *features* porque já foi estudado que usar a Acurácia não é muito eficiente para o contexto dos dados deste trabalho, assim como mostra os resultados do Capítulo 5. RMSE é uma medida que descreve a distância quadrática entre o valor estimado e o real valor da classificação, então quanto menor a métrica, mais perto de classificar corretamente está o modelo. Os gráficos presentes na Figura 28 mostram os resultados em cada projeto de como a métrica RMSE se altera em função da quantidade das melhores *features* selecionadas.

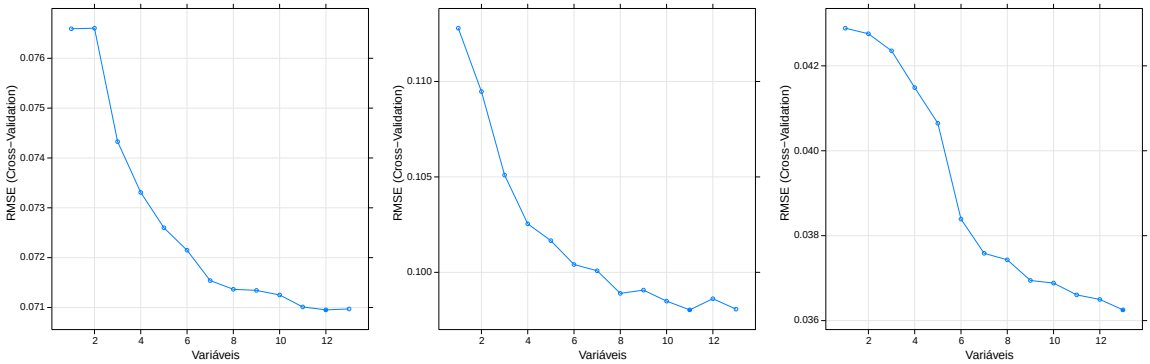
Observando as imagens percebe-se que os resultados foram distintos para cada projeto, isso se dá por vários fatores, desde a quantidade de vulnerabilidades colhidas ao

Figura 28 – RMSE em função da quantidade de *features* selecionadas.

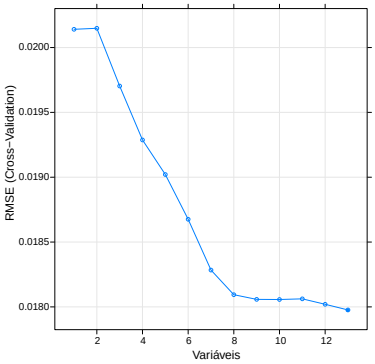
(a) Xen HV (b) Glibc (c) Httpd



(d) Derby (e) Tomcat (f) Kernel



(g) Mozilla



Fonte: elaborada pelo autor.

contexto de cada projeto. Observando de uma maneira geral, percebe-se que os resultados tenderam a acolher um maior número de métricas e que esse aumento melhorava a predição, com exceção apenas do Xen HV que teve uma oscilação muito grande a medida que se acrescentavam *features*.

Os dados do projeto Xen HV tiveram os resultados com maior oscilação, mas mesmo assim, foram selecionadas 10 métricas como bons descritores de vulnerabilidades. Como resultado fora da normalidade dos outros projetos estudados, quando se acrescentava a

decima primeira e decima segunda métrica, a métrica RMSE aumentava consideravelmente em relação as outras, outros pontos de oscilação são percebidos com a adição da terceira e sexta métrica. Essa oscilação pode ser causada pelo fato de conter métricas que isoladamente não pareciam bons preditores, mas com o acréscimo de nova informação (uma nova métrica) essa característica pode mudar.

Glibc e Httpd são dois projetos escritos em C, em que pegou-se as menores amostras de vulnerabilidades. Teve muita variação na seleção das melhores características, como podem observar nas Figuras 28b e 28c, a linha chega a fazer zigue-zague, entretanto, as variações da métrica RMSE variam em uma determinada amplitude. Isso mostra que apesar de ter escolhido certo ponto de menor erro, as características mostram que essas métricas podem servir de preditores, mas que nessas amostras, não teve boa força preditiva.

Derby e Tomcat são dois projetos escritos na linguagem Java e que tiveram resultados bem interessantes. Apesar da quantidade de vulnerabilidades colhidas não superarem muito os outros projetos (Glibc e Httpd), as linhas das Figuras 28d e 28e tiveram uma diminuição constante a medida que se adicionavam as características novas. Isso mostra que as métricas estão auxiliando na medida em que são utilizadas para realizar predição. No caso do Derby foram escolhidas 12 métricas, eliminando apenas uma, mas que se observar essa última se fosse adicionada a taxa de erro quase não diminuiria, e no caso do Tomcat, foram selecionadas 11, seguindo o mesmo comportamento das métricas não selecionadas no Derby.

Quando observam-se os gráficos das amostras do Mozilla e do Kernel, percebe-se que esses seguem melhorando o resultado à medida que foram adicionada novas *features*. Nesses dois casos, todas as métricas foram selecionadas, mostrando assim que cada uma delas acrescentou informação e diminuiu o erro quadrático RMSE. Essas duas amostras são extremamente importantes pois são projetos de muita maturidade, tempo de desenvolvimento e foi possível coletar uma boa quantidade de vulnerabilidades, sendo eles os que detém a maioria dos dados no *dataset* construído.

Para mostrar de fato quais métricas foram escolhidas como os melhores preditores em cada projeto, foi elaborada a Tabela 13 marcada com um "X"nas métricas selecionadas em cada amostra de projeto.

De fato não era esperado que a quantidade de atributos fosse pouco reduzida, na maioria dos projetos foram escolhidas mais de nove atributos, com exceção do Glibc com apenas 5. Isso difere de alguns modelos já existentes, como mostrado no Capítulo 5, que em alguns casos pega os três melhores atributos, já outro pega os sete melhores atributos.

Como conclusão, é possível dizer que todas as métricas serviram com informações relevantes para predição de vulnerabilidades, entretanto, algumas tiveram resultados positivos com unanimidade, são elas: *CountLineBlank*, *CyclomaticStrict*, *Knots* e *CountOutput*.

Tabela 13 – Métricas selecionadas como melhores preditores em cada amostra.

Métricas	Mozilla	Kernel	Xen HV	Httpd	Glibc	Tomcat	Derby
CountLine	X	X	X			X	X
CountLineCodeDecl	X	X	X			X	X
CountLineBlank	X	X	X	X	X	X	X
CountStmt	X	X	X	X		X	X
CountStmtDecl	X	X	X			X	X
CyclomaticStrict	X	X	X	X	X	X	X
MaxNesting	X	X		X			
Knots	X	X	X	X	X	X	X
MaxEssentialKnots	X	X		X		X	X
CountPath	X	X	X				X
RatioCommentToCode	X	X		X	X	X	X
CountInput	X	X	X	X		X	X
CountOutput	X	X	X	X	X	X	X

Fonte: elaborada pelo autor.

6.3 PCA

Como segunda opção, foi utilizada uma técnica de redução de dimensionalidade da amostra para verificar quais atributos tem maior influência na variância da amostra. Para isso utilizou-se o PCA, acrônimo para Análise de Componentes Principais, que tem por finalidade encontrar um pequeno número de combinações lineares das variáveis de modo a capturar a maior parte da variação no quadro de dados como um todo (CRAWLEY, 2012).

Para aplicar essa técnica foi utilizado um algoritmo na linguagem R denominado *prcomp*. Ao aplicar o PCA nas amostras de cada projeto, foi verificada a variância acumulativa de cada uma delas, como mostra a Tabela 14:

Tabela 14 – Proporção da variância cumulativa em cada projeto.

Variância (%)	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10
Mozilla	0,46	0,60	0,69	0,77	0,84	**0,90**	0,94	0,97	0,98	0,99
Kernel	0,34	0,47	0,56	0,64	0,71	0,78	0,85	**0,91**	0,94	0,96
Xen HV	0,58	0,72	0,80	0,88	**0,93**	0,97	0,98	0,99	1,00	1,00
Httpd	0,54	0,68	0,76	0,83	0,89	**0,93**	0,96	0,97	0,99	0,99
Glibc	0,37	0,51	0,61	0,70	0,78	0,83	0,89	**0,93**	0,96	0,98
Tomcat	0,49	0,64	0,73	0,80	0,86	**0,92**	0,95	0,97	0,98	0,99
Derby	0,36	0,50	0,60	0,68	0,75	0,82	0,88	**0,93**	0,96	0,98

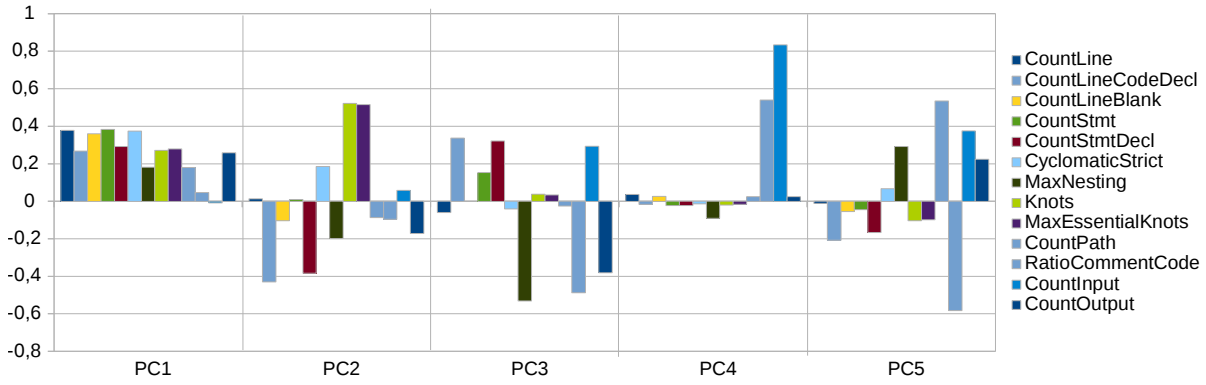
Fonte: elaborada pelo autor.

Para selecionar os Componentes Principais com maior importância, foi adotada uma prática prática padrão de seleção de 90% da variância (CRAWLEY, 2012). Os pontos em negrito na Tabela 14 representam até que componente a variância cumulativa atinge o valor de 90%, que nesse caso, a maior parte da variância está majoritariamente composta entre 5 e

8 componentes, num total de 13 componentes criados pelo algoritmo. Foram selecionados apenas os 10 com maior variância pela relevância da informação, os demais aumenta muito pouco a variância e nem serão selecionados para estudo, assim como não mostram nenhum ponto crítico, ou limiar para investigação do trabalho no contexto aqui estudado.

Para avaliar os pesos atribuídos a cada *feature*, foi verificado a matriz de rotação de cada combinação linear dos componentes escolhidos. Um exemplo para ser mostrado é retirado da amostra do Mozilla que é apresentado na Figura 29.

Figura 29 – Pesos de cada atributo em cada Componente Principal



Fonte: elaborada pelo autor.

Como podem observar na Tabela 14, o PC1 do Mozilla apresenta a maior variância entre os componentes detendo 46% de toda a variância de sua amostra, o que mostra sua devida importância. Ao observar os pesos atribuídos a cada métrica nesse componente, percebeu-se que as métricas *RatioCommentToCode* e *CountInput* foram as que receberam menor valores e são pouco relacionadas a esse componente. Comparadas com os outros componentes, essas métricas não influenciam tanto, aparecendo consistentemente apenas nos últimos componentes que apresentam pouca variância da amostra.

Outros projetos foram avaliados e percebeu-se que para o Kernel e Xen HV, o mesmo acontecia com as métricas *RatioCommentToCode* e *CountInput*. Para o Glibc, os pesos de *CountInput*, *CyclomaticStrict* e *RatioCommentToCode* estavam os mais próximos de 0, entretanto, foram usados na criação de outros componentes com menor variação da amostra. O Httpd teve os valores dos autovetores negativos, o que mostra um componente inversamente proporcional ao problema tratado, e teve como piores métricas: *RatioCommentToCode* e *CountInput*, só sendo consistentemente usadas no PC3 que detém apenas 8% da variação total da amostra, algumas outras métricas como *Knots* e *MaxEssentialKnots* tiveram resultados inferiores, mas foram bem usadas no PC2, apesar de no restante terem resultados pequenos comparado com as demais métricas. Tomcat e Derby obtiveram semelhanças com os resultados do Mozilla, tendo como piores índices as métricas *CountInput* e *RatioCommentToCode*, mais além, o Derby apresentou resultados ruins para a métrica *Knots*

também, só sendo utilizadas em Componentes com pouca variação da amostra.

Como conclusão dos resultados do PCA, notou-se que todas as métricas foram utilizadas para criação dos componentes, como esperado, mas algumas tiveram pesos superiores as outras e com redução drástica em algumas métricas como *CountInput* e *RatioCommentToCode*. Já as demais tiveram resultados positivamente equilibrados, com oscilações nas métricas *Knots* e *MaxEssencialKnots* e *CyclomaticSrict*.

6.4 CONCLUSÃO

Neste capítulo foi realizado um estudo para seleção das melhores métricas utilizando duas abordagens. Inesperadamente, os resultados cobriram a maioria das métricas ou quase todas, resultando em ganho de informação para o uso de todas as métricas selecionadas após a simplificação de atributos redundantes. Esse resultado reforça ainda mais que essas métricas tem certo poder discriminativo para funções vulneráveis. É notório que algumas tem maior poder discriminativo que outras, por isso alguns resultados tiveram oscilações e poderiam seleciona-las a partir de um *rank* para maior agilidade no desempenho dos modelos, como forma de simplificar o conjunto de dados e retirar as menos importantes, mas este método não seria tão viável quanto parece, visto que eliminaria informações que já foram julgadas pelas técnicas como importantes e se os resultados forem comparados por meio de um *rankeamento*, percebe-se que para o RFE, a métrica *CountImput* está entre as melhores e para o PCA está com menos pesos para os auto vetores dos componentes.

7 AMEAÇAS A VALIDADE

Neste capítulo será discutido sobre as ameaças a validade do trabalho, separados em categorias referentes a tipologia da ameaça seguindo as divisões descritas no livro de Wohlin (WOHLIN et al., 2012).

7.1 AMEAÇAS A CONCLUSÃO

Uma ameaça está na extração da informação nos *reports* de vulnerabilidades, apesar de ter escolhido esse ambiente como fonte de conhecimento, ela ainda apresenta algumas dificuldades, visto que muitos relatórios de segurança não são uniformes e não seguem um padrão. Por esse motivo, uma parte deles não foi coletado, cerca de 48% das ameaças de segurança não foram colhidas por motivos como:

- A linguagem estava fora do escopo do trabalho, por exemplo, a correção era em código JavaScript, ou em arquivos HTML;
- Não conter informações de reparos em alguns alertas de segurança;
- Não existir informações referentes ao repositório de código e/ou não foi possível identificar o *commit* responsável.

Desta forma, corre-se o risco de apresentar heterogeneidade randômica da amostra, visto que aparecem várias brechas de captura de amostras de vulnerabilidades durante esse processo, mas apesar disso, foram colhidas o maior número de amostras possível, tendo muitos exemplos para estudo.

7.2 AMEAÇAS EXTERNAS A VALIDADE

Outra ameaça a validade deste trabalho está diretamente relacionado a base de dados criada. Os dados foram retirados de vulnerabilidades reportadas em relatórios de segurança. Isso especifica os dados a apenas uma fonte de conhecimento, pois seria possível retirar informações de ferramentas de detecção e outras fontes. Entretanto, mitigar informações relativas a essa de outras fontes, requer mais custo e processamento. Por exemplo, para ter o objetivo de adicionar informações relativas a vulnerabilidades de ferramentas de detecção, é preciso encontrar quais ferramentes teriam maior precisão e acurácia e assim realizar a detecção em cada *snapshot* ou até mesmo em cada *commit* de código, o que já seria muito custoso. Além disso, haveria de ser feito validações manuais em cada vulnerabilidade detectada, visto que as ferramentas atuais não detectam vulnerabilidades com precisão

satisfatória, seria preciso a opinião de especialistas e desenvolvedores para validação da amostra coletada. Visto isso, decidiu-se seguir a metodologia de algumas pesquisas, e expandir a coleta de informação para uma quantidade maior de dados coletados.

Em relação a reconstrução dos ambientes de modelos preditivos comparados, há uma ameaça na replicação dos modelos de predição presentes na literatura atual. A comparação desses modelos seguiu os parâmetros de acordo com as informações colhidas de cada estudo (artigo), entretanto elas ainda poderiam ser melhor ajustadas, mas devido à falta de informação, foram utilizadas as configurações padrão do Weka.

7.3 AMEAÇAS DE GENERALIZAÇÃO DOS RESULTADOS

A próxima ameaça é referente aos projetos escolhidos. As observações encontradas podem não ser generalizados para outros projetos, no entanto, percebe-se que os projetos selecionados são bem representativos e são projetos amplamente expostos a problemas de segurança. A ideia foi diversificar ao máximo a escolha dos projetos, tendo projetos escritos apenas em C, outros em C/C++ e outros escritos na linguagem Java. Também foram selecionados projetos com contextos e tamanhos diferentes, o que pode ser crucial para entender como os resultados se comportam em cada domínio estudado.

8 TRABALHOS RELACIONADOS

Neste capítulo serão descritos alguns trabalhos relacionados à pesquisa realizada neste. Serão apresentados vários trabalhos distintos, tanto os que foram abordados na seção 5 da análise comparativa, quanto outros com contextos diferentes mas relacionados ao trabalho aqui realizado.

A Tabela 15 apresenta a relação dos trabalhos relacionados com os aspectos que eles abordam em seu trabalho. Como podem observar, alguns trabalhos podem estar associados a mais de um contexto, dependendo da aplicabilidade desse contexto nos trabalhos referenciados.

Tabela 15 – Contextos dos trabalhos relacionados.

Trabalhos Relacionados	**Predição de Falhas**	**Construção de Banco**	**Relação Métrica e Vulnerabilidade**	**Predição de Vulnerabilidade**	**Redução de Dimensionalidade**
(SHIN et al., 2012)	X	X			
(CATAL; DIRI, 2009a)	X				
(ABAEI; SELAMAT, 2014)	X				X
(SHIN; WILLIAMS, 2013)	X			X	
(CATAL; SEVIM; DIRI, 2011)	X				
(CATAL; DIRI, 2009b)	X				
(OHIRA et al., 2015)		X			
(HOWARD et al., 2013)		X			
(DAVARI; ZULKERNINE, 2016)		X		X	
(SHIN et al., 2011)			X	X	X
(MCGRAW, 2006)			X		
(SHIN, 2008)				X	
(CHOWDHURY; ZULKERNINE, 2011)			X	X	
(WALDEN; STUCKMAN; SCANDARIATO, 2014)				X	
(SMITH; SHIN; WILLIAMS, 2008)			X	X	
(MENEELY et al., 2014)				X	
(SHIN; WILLIAMS, 2011)				X	
(STUCKMAN; WALDEN; SCANDARIATO, 2017)				X	X
(MENZIES; GREENWALD; FRANK, 2007)				X	X
Este trabalho		X	X	X	X

Fonte: elaborada pelo autor.

Este capítulo estará dividido em cinco seções da mesma forma que está separado as colunas da Tabela 15. A primeira são trabalhos referentes a predição de falhas, ao qual são pesquisas base para espelhar-se e mais maduras, que pretensiosamente a predição de vulnerabilidades chegará um dia. A segunda focando nos aspectos relacionados a construção de conjuntos de dados, destacando-se trabalhos com produção, técnicas e desafios na mineração de informações em repositórios. A terceira está direcionado a relação entre métricas de software e vulnerabilidades, com foco na discussão de hipóteses e *insights* iniciais para investigações nesse contexto. A quarta são trabalhos de predição de vulnerabilidades atuais e a quinta são trabalhos relativos a redução de dimensionalidade e seleção de *features* ao qual também foi realizada atividade semelhante em nossa pesquisa.

8.1 TRABALHOS DE PREDIÇÃO DE FALHAS

A existência de várias pesquisas criando modelos de predição de vulnerabilidades são influenciadas também por pesquisas que criam outros modelos preditivos, como por exemplo, modelos de predição de falhas. Esses modelos são mais maduros por terem um tempo de investigação maior, além de outros fatores. Por esse motivo será discutido a respeito de algumas pesquisas que podem servir de inspiração e base para realização de abordagens parecidas quando trata-se de vulnerabilidades de software.

Por exemplo, em (SHIN et al., 2012) é utilizado diversos atributos para criação dos modelos, entretanto, um pouco diferentes das métricas de software estudadas neste trabalho. Nele, o estudo foi voltado a estruturas de chamadas, apesar das outras métricas. Uma estrutura de chamadas representa a relação de invocação entre métodos de um programa. Seus resultados não mostraram melhorias nos novos modelos, entretanto, há métricas que podem trazer informações similares e que em vulnerabilidades se destacaram. As métricas *CountInput* e *CountOutput* não contam especificamente as chamadas externas, mas contam as chamadas internas nos dando informações implicitamente similares, e para nossos resultados, essas métricas se destacaram em algumas de nossas observações.

Outro ponto destacado e de grande importância é o fator do tamanho e contexto da amostra. Isso foi discutido em (CATAL; DIRI, 2009a), que avalia a influência do tamanho da amostra e redução de *features* sobre os algoritmos de aprendizagem de máquina em modelos de predição de falhas. Seus experimentos demonstraram que *random forest* é melhor quando aplicado em conjunto de dados grandes enquanto *naive bayes* é melhor aplicado em conjuntos de dados menores. Entretanto, essa distinção é em relação a diminuição da dimensionalidade da amostra, seria interessante investigar projetos com contexto e tamanhos de código diferentes, assim como os dados desta pesquisa permitem fazer.

Um outro trabalho compara várias abordagens de predição de falhas (ABAEI; SELAMAT, 2014), desta forma, investiga a utilização de vários algoritmos de aprendizagem de máquina aplicados a conjuntos de dados públicos disponibilizados pela NASA. Esse é o tipo de estudo que poderia ser realizado na área de vulnerabilidades, entretanto com mais instâncias, métricas e amostra de dados, tendo sempre que solucionar os problemas específicos desse contexto (balanceamento e métricas de medição). Esse trabalho se limitou a comparar diversos algoritmos de aprendizagem de máquina, entretanto como já dito, outros algoritmos referentes a balanceamento e dimensionalidade da amostra devem ser observados, principalmente se o contexto for modelos de predição de vulnerabilidades.

Em (SHIN; WILLIAMS, 2013) foi realizado um trabalho em que avaliam o desempenho de predição baseado em um modelo de predição de falhas de software, em particular, uma vulnerabilidade é uma falha de software, entretanto são delicadamente diferentes e o inverso nem sempre é verdade. Nesse trabalho, utilizam quinze métricas de software para realizar

as predições e treinamento dos algoritmos, dentre as quais algumas delas são métricas de mudança de código. Essas métricas estão relacionadas aos *commits* realizados no arquivo. Nesse trabalho foram realizados vários experimentos e foi constatado que não é possível se chegar a prever vulnerabilidade de software a partir de um modelo de falhas, isso acontece porque a quantidade de vulnerabilidades é bem inferior a quantidade de falhas no software. A base de dados utilizada foi retirada do Mozilla Firefox 2.0 web browser, o que restringe ainda mais a pesquisa, apesar de bem elaborada. Para se chegar a conclusões mais concisas é preciso diversificar ainda mais a quantidade de projetos e de amostras de vulnerabilidades.

Além das pesquisas, é preciso que todos esses trabalhos tenham aplicabilidade. Em (CATAL; SEVIM; DIRI, 2011) é construída uma ferramenta denominada *RUBY Plugin* que funciona junto ao eclipse e que simplifica o processo de predição de falhas treinando um algoritmo de aprendizagem de máquina conhecido como *naive bayes*. O ponto fraco dessa ferramenta é que o desenvolvedor deverá colocar conhecimento e calibrar o algoritmo antes dele gerar resultados, apesar de não pensar em alternativa melhor. Apesar disso, tal abordagem poderia ser utilizada para o contexto de vulnerabilidades, pois é difícil pensar em ferramentas para realização dessa tarefa de predição de maneira automática, sendo esse um bom destaque para investigações na área de falhas de software que surgem em escassez no contexto de vulnerabilidades. Para sanar a calibração manual do desenvolvedor, poderia se adicionar conhecimento inicial pelo qual pudesse vir de métricas que representam códigos com altas chances de ter vulnerabilidades, mas esse é um estudo que ainda não se chegou a resultados interessantes, entretanto a cada ano existem avanços nessa área.

É notável que pesquisas relacionada a predição de falhas são mais maduras que as relacionadas as vulnerabilidades de software. Isso se deve a vários fatores, desde a facilidade de encontrar amostras à gravidade imediata que os *bugs* podem causar. Em (CATAL; DIRI, 2009b) é realizado uma revisão sistemática sobre as pesquisas relacionadas às métricas, aos conjunto de dados e à predição de falhas, fatos que podem ser relevantes ao se predizer vulnerabilidades. Primeiro que as métricas relacionadas a métodos ainda são mais relevantes na predição de falhas em várias pesquisas, outro que complexidade ainda é fator de notória importância na discriminação de falhas e por fim, várias abordagens foram exploradas e que isso foi de notória importância para os avanços dos resultados investidos nessa área. Entretanto apesar desses fatores estarem em crescimento na área de vulnerabilidades de software, foi possível coletar uma amostra considerável de vulnerabilidades nos projetos, o que ajuda a fortalecer ou enfraquecer os resultados atuais das pesquisas nessa área.

8.2 METODOLOGIAS DE CONSTRUÇÃO DE BANCO DE DADOS

Assim como fizemos, muitos outros trabalhos seguem a mesma linha de produção de grandes conjuntos de dados e mineração de informações em repositórios (OHIRA et al., 2015; HOWARD et al., 2013; GOUSIOS; ZAIDMAN, 2014; SPINELLIS, 2015).

Em (OHIRA et al., 2015) é realizado uma mineração de dados nas *issues* reportadas de quatro projetos *open source*: Ambari, Camel, Derby e Wicket. Esses dados foram filtrados com uso de palavras chaves nos repositórios, mas além disso, realizaram uma classificação manual de quatro mil *issues* classificando-os em relação ao desempenho, segurança, bloqueio e outras classificações. Tanto que foi possível aproveitar os dados dos *bugs* do Derby classificados como influentes no fator segurança. Essa foi uma contribuição consistente para comunidade, mas esse não é o melhor modelo de coleta de informações sobre *bugs*, pois demanda muito tempo e exige especialistas para validação manual, isso porque projetos atuais já publicam em repositórios específicos para esse fim os *bugs* dos projetos.

Já que a pesquisa por palavras chave foi um fator de importância no filtro de *issues* em (OHIRA et al., 2015), a ferramenta de Matthew J. Howard (HOWARD et al., 2013) poderia ser base para aplicação da mesma eurística de similaridade de palavras para identificar *bugs* e vulnerabilidades em repositórios de código e *issues*, só que ao invés de aplicar em comentário de código, poderia ser aplicada em descrições de *commits* para se identificar padrões. Essa abordagem não foi adotada porque focou-se em repositórios de vulnerabilidades, entretanto para projetos que não tem esse tipo de repositório. Essa abordagem poderia ser aplicada em repositórios de código com o intuito de adicionar ainda mais exemplos de vulnerabilidades nos bancos de dados.

Outros trabalhos focaram apenas em repositórios de vulnerabilidades, como em (DAVARI; ZULKERNINE, 2016), que investiga o Firefox browser, expondo as dificuldades e desafios de se construir e trabalhar com o contexto de vulnerabilidades de software coletando dados desses repositórios. Nele, Davari utiliza o repositório do Mozilla para conduzir sua pesquisa, e também utiliza a ferramenta Understand para coletar as métricas, mas trazendo uma separação distinta das métricas. Visto seus resultados, indicando que algumas métricas não contribuem para ocorrências de vulnerabilidades, foi percebido que alguns fatores não foram observados, e fatores que implicam diretamente em suas conclusões, como: as métricas de medição que se encaixam melhor no contexto de vulnerabilidades (*informedness* e *markedness*) e problemas de balanceamento dos dados, que mesmo utilizando de técnicas de balanceamento, a discrepância é tão grande que os algoritmos se tornam tendenciosos.

8.3 ESTUDO DA RELAÇÃO ENTRE MÉTRICAS E VULNERABILIDADE

Pesquisas estudam a relação entre métricas de software e vulnerabilidades, mas poucas se destacam na criação de hipóteses para explicitar de maneira clara como essas métricas de software realmente discriminam vulnerabilidades. Muitas pesquisas normalmente assumem que essa relação existe estatisticamente mas não traz argumentação suficiente para mostrar a causalidade entre essas partes.

Em (SHIN et al., 2011), criam-se diversas hipóteses baseadas em cada tipo de métrica e como elas poderiam ser a causa da existência de vulnerabilidades. Não explicita cada métrica, mas fala de tipos básicos de métricas como determinantes para discriminar vulnerabilidades. Por exemplo, ele afirma que o código altamente acoplado tem maior chance de ter entradas de fontes externas que são difíceis de rastrear de onde a entrada veio. Até em relação a comentários de código que ajudam novatos quando a equipe é grande, mas que códigos feitos à pressa e com poucas preocupações podem ter menos comentários. Para além disso, ele ainda cria hipóteses para métricas de mudança de código, quando afirma que um código mudado e alterados várias vezes durante sua vida útil, tende a ter mais problemas, isso porque é provável que o desenvolvedor venha a esquecer certos comportamentos do código, e atividade do desenvolvedor, quando ele afirma que um código alterado por várias pessoas diferentes tem chance de conflitos e ter comportamentos adversos, pois cada pessoa tem raciocínios diferentes. É esse tipo de argumentação que falta em muitas pesquisas sobre vulnerabilidades de software, entretanto mais aprofundada. O ponto aqui é que ele afirma que 24 dentre 28 métricas tiveram resultados positivos para discriminar vulnerabilidades sendo que existem métricas que discriminam e outras que não se baseiam na mesma hipótese. O fato é que nem todas as métricas de complexidade que representam o acoplamento serão uteis para discriminar vulnerabilidades, então tal hipótese deve ser reformulada sendo especificada melhor.

O mesmo é discutido em (MCGRAW, 2006), entretanto de uma maneira geral. McGraw afirma que a complexidade tem uma tendência forte no impacto a segurança do software. Fatores como acoplamento, que podem ser discriminados com métricas, sobreposição de reparos, correções e erros arquiteturais, podem ser significantes para segurança de um sistema. Ainda mais quando a linguagem é *unsafe programing language* [1] permitindo ainda mais que programadores deixem escapar uma brecha pelo fato dessas linguagens não proteger a simples tipos de ataque como *buffer overflows*. O fato é que suas observações sobre complexidade são válidas, mas por serem gerais, não ajudam tanto para aqueles que querem diminuir a quantidade de vulnerabilidades em seus programas de maneira prática, mas mesmo assim, serve de gatilho para investida em pesquisas com maior profundidade.

O fato é que muitas pesquisas não demonstram como essas métricas foram escolhidas e nem por qual motivo elas conseguem, independentemente, discriminar vulnerabilidades. O que se tem são hipótese formuladas de maneira geral e que expressam uma pseudo-causalidade entre complexidade estrutural e vulnerabilidades de software. Na seção 9.3 foram selecionadas as melhores métricas a partir dos nossos experimentos e discutido um pouco mais profundamente cada uma delas, sugerindo estudos mais aprofundados sobre como elas podem ser melhor estudas.

[1] O programador tem que alocar quantidades específicas de espaço de memória e permissão para manipular ponteiros que gerenciam endereçamentos de memória.

8.4 TRABALHOS DE PREDIÇÃO DE VULNERABILIDADES

Existem diversos trabalhos no estado da arte no quesito de predição de vulnerabilidade, entretanto essa área ainda sofre com alguns problemas que prejudicam uma boa análise e modelagem de preditores, que é a pequena quantidade de vulnerabilidades reportadas. Por esse motivo, muitos trabalhos utilizam projetos diferentes, gerando problemas já citados anteriormente, cada um com métricas e técnicas particulares.

No trabalho de Y.Shin (SHIN, 2008) são utilizados nove métricas de complexidade estrutural, extraídas de um módulo do Mozilla Firefox em um determinado período de tempo. Seu trabalho consistiu em responder algumas questões de pesquisas e determinar a possibilidade de predição de vulnerabilidades em artefatos de código, mas os resultados geraram muitos falsos negativos e se determinou esse o objetivo para ser alcançado em trabalhos futuros. Entretanto utilizaram uma abordagem baseada em lançamentos de versões, e nossa abordagem é mais contínua. Outro ponto é a utilização de dados bem restritos, o módulo de motor JavaScript do Mozilla pode seguir padrões específicos e acabar por especificar os resultados apenas para esse pequeno domínio.

No trabalho de I. Chowdhury (CHOWDHURY; ZULKERNINE, 2011) foram utilizadas dezessete métricas de software e sua base de dados é retirada do Mozilla, onde fizeram uma coleta de dados de cinquenta e duas versões do projeto. Nesse trabalho o autor faz uma comparação de modelos de predição avaliando seus desempenhos. Seu trabalho também é baseado em lançamentos de versões do software. Entretanto, ainda há limitações como a quantidade de projetos observados e a granularidade da abordagem, que é a nível de arquivo. Seus resultados, no geral, apresentaram-se como razoáveis, chegando a prever 75% dos arquivos com vulnerabilidades, com uma porcentagem de menos de 30% de falsos positivos e acurácia de 74%.

Alguns trabalhos investigam a possibilidade de técnicas diferentes para predição de vulnerabilidade, como exemplo *Text Mining* (HOVSEPYAN et al., 2012; SCANDARIATO et al., 2014; WALDEN; STUCKMAN; SCANDARIATO, 2014). Walden em seu trabalho (WALDEN; STUCKMAN; SCANDARIATO, 2014) faz uma comparação do uso de duas técnicas para realização da predição de vulnerabilidade. Ele compara o uso de métricas de software em relação ao uso de *Text Mining* para realizar a predição. Ao avaliar o desempenho das duas abordagens sobre um conjunto de dados formados pela coleta de dados de três projetos escritos em php - Drupal, Moodle e PHPMyAdmin - obtiveram como resultado um desempenho melhor utilizando a técnica de *Text Mining* para predição, entretanto, algumas limitações devem ser observadas. O autor realiza a comparação, na abordagem de métricas de software, utilizando doze métricas de software e as utiliza de forma genérica, visto que algumas métricas podem não ser eficientes para avaliar determinadas vulnerabilidades, além do que a quantidade de arquivos vulneráveis encontrada é muito pequena. Foi feito um levantamento das métricas relatadas pelos trabalhos e serão concluídas quais são realmente

as mais correlacionadas às vulnerabilidades nos próximos capítulos.

No decorrer deste trabalho apresentou-se uma análise detalhada sobre as métricas de software já levantadas e distinguiu-se quais delas seriam mais adequadas para identificar determinadas vulnerabilidades. Assim como no trabalho de Smith (SMITH; SHIN; WILLIAMS, 2008) que propôs duas métricas que seriam eficientes para identificar, por exemplo, *SQL Injection*. Em sua coleta de dados, foram utilizadas informações de um projeto chamado iTrust e chegou-se a resultados interessantes com 96.7% de cobertura. Entretanto ele reforça que o projeto iTrust poderia ter facilitado a predição por decorrentes fatores como uma estruturação bem definida das funcionalidades.

Em (SHIN et al., 2011) foi realizado um estudo em que avaliavam o desempenho de 28 métricas de software que se apresentavam muito relevante porque utilizaram métricas de mudança de código (*code churn*), complexidade estrutural e a atividade do desenvolvedor durante os *commits* de código. A base de dados foi construída a partir de repositórios de *bug* e histórico de vulnerabilidades, utilizando dois projetos para coleta de informação, Mozilla Firefox com vulnerabilidades reportadas no período de janeiro de 2005 à agosto de 2008 e O Red Hat Enterprise Linux Kernel coletando vulnerabilidades no período de fevereiro de 2005 à julho de 2008. Apesar de uma metodologia interessante, usaram informações muito antigas de uma base de dados dos projetos. Por esse motivo, algumas dessas vulnerabilidades já deixaram de ser meios intensos de foco de ataques e devendo-se acrescentar vulnerabilidades mais recentes. Outro quesito é em relação a abordagem que está a nível de arquivo, o que limita a inspeção de código. Arquivos podem ter mais de cinco mil linhas de código, o que pode ser demasiado custoso para o desenvolvedor analisar, além de que serão selecionados os que apresentam maior complexidade e volume.

Em um outro estudo (MENEELY et al., 2014) os autores investigam a possibilidade de precisão a partir de métricas relacionadas a revisão de código. Sua base de dados é retirada do projeto *open source* Chromium e contou dez métricas de software. Teve resultados positivos para realização da predição, entretanto mais investigações são necessárias, pois os resultados mostram que há uma correlação, mas não demonstrou como essas relações funcionam, por isso é necessário uma análise estatística mais aprofundada.

Como sugestão para melhorar a predição, (SHIN; WILLIAMS, 2011) realiza um estudo inicial investigando a possibilidade de se utilizar métricas de execução para realizar a predição de vulnerabilidades. Nesse estudo o autor realiza a coleta de dados de dois projetos, Mozilla e Wireshark e concluiu que essas métricas são discriminativas para a predição dos dados do Mozilla, entretanto o mesmo não acontece para o Wireshark, assim precisando de mais estudos para investigar esse fato.

8.5 TRABALHOS DE REDUÇÃO DE DIMENSIONALIDADE

A redução de dimensionalidade e limpeza dos dados é um procedimento indispensável para o treinamento de algoritmos e avaliação do desempenho de modelos de predição para qualquer finalidade. Vários são os estudos para melhorar o desempenho desses modelos. Em nosso caso, o foco se concentrou nos trabalhos que se adaptavam melhor ao nosso problema com as vulnerabilidades, destacando assim este trabalho a seguir.

Em (STUCKMAN; WALDEN; SCANDARIATO, 2017), é realizado um estudo sobre como a redução de dimensionalidade de amostras de instâncias vulneráveis e neutras pode melhorar o desempenho de modelos preditivos. Nesse caso o autor utiliza modelos baseados em métricas de software e *text mining*, assim como cinco métodos de redução de dimensionalidade, dois deles estudados neste trabalho para destacar as melhores métricas. Foi utilizado com essa finalidade porque nosso objetivo não é melhorar modelos de predição ou realizar tal comparação de técnicas, e sim mostrar, em um nível mais baixo que essas métricas tem relações próximas com artefatos vulneráveis. Como resultados, para métricas de software, eles tiveram uma melhora nos resultados, mas para *text mining* os resultados só trouxeram ganho de processamento computacional.

Os resultados interessantes que obtive-se na seleção de métricas, tiveram como base o resultado de uma pesquisa de Menzies (MENZIES; GREENWALD; FRANK, 2007), em que se critica a forma de comparações entre tipos de métricas e parte para união de todos os contextos de métricas para construção de preditores. Em seu estudo, o uso de todos os tipos de métricas contribuiu para melhora dos modelos preditivos, apesar de algumas ressalvas de variações de projeto para projeto. Em nosso estudo, além de defender esse tipo de argumentação, nos coube relacionar e defender as métricas dentro de cada tipo, onde existiram resultados em que três métricas de contagem de linhas de código foram relevantes e distintas, assim como duas de contagem de chamadas externas e assim por diante, como mostrou o Capítulo 6.

9 LIÇÕES APRENDIDAS

Neste capítulo é discorrido sobre as atividades, métodos e condições para realização deste trabalho, comentando suas eficiências e utilidades. Técnicas que deram certo e as que falharam, assim como apresentar as razões porquê isso aconteceu e as lições que foram aprendidas, resumindo assim as barreiras e oportunidades deste trabalho.

9.1 DESAFIOS E BARREIRAS DA COLETA DE DADOS

Para realizar a coleta de dados deste trabalho, foram utilizadas técnicas de extração de dados sobre repositórios de vulnerabilidades reportadas. A primeira vista nos pareceu uma ótima ideia, mas com o avanço na pesquisa, percebeu-se que cada vez que eram incluídos novos projetos ou adicionados dados, tinha-se que refazer todo um trabalho exaustivo. O principal problema desse quesito é que os dados, muitas vezes, fogem do padrão e assim a técnica de *web scrapping* falhava, forçando-nos a readaptar o código de coleta de informação automática.

Um dos grandes desafios nessa pesquisa foi identificar os *commits* responsáveis pela correção da vulnerabilidade. Isso porque muitas vezes essa informação não aparecia nos relatórios de vulnerabilidades do repositório, então foi preciso criar um *script* que, dados os trechos de código da correção e o repositório de código do projeto, era possível encontrar os *commits* que realizaram aquelas mudanças corretivas. Essa tarefa foi uma das mais críticas para os projetos do Mozilla, Xen e Derby, e por isso ainda foi preciso realizar uma validação manual nos dados para verificar a veracidade da informação, eliminando as que apresentavam informações erradas.

Ainda na construção do banco de dados, outro grande desafio foi realizar a coleta das funções vulneráveis. No primeiro momento, tentou-se realizar essa coleta nos cabeçalhos dos repositórios de código, mas logo foram encontrados erros e informações inconsistentes com os *commits* apresentados. Isso acontecia porque eles seguiam uma metodologia fixa para mostrar o *diff* de código e não se importavam com casos fora do padrão. Cientes disso, o código e a coleta da informação foi adaptada e feita do próprio repositório de código, criando a árvore sintática do código em determinado *commit* e coletando a assinatura do método ou função.

Em relação ao banco de dados, houve diversas barreiras com armazenamento dos dados, primeiramente que armazena-los de forma contínua e colocando todas as informações de cada *commit* do repositório de código se tornaria inviável pelo grande volume de informações. Então, essas coletas foram realizadas apenas nos reparos em que se ocorria uma correção de vulnerabilidades. Mesmo assim chegou-se a 2.361 reparos, que refletiu 4.722

snapshots de código para se extrair informações de todos os métodos, classes e arquivos. Mesmo assim, a quantidade de registros ainda era um problema e para solucionar isso, as tuplas duplicadas foram eliminadas só armazenando uma informação temporal sobre a tupla, o que solucionou o problema, mas ainda deixa inviável a obtenção da base em alguns computadores pessoais com pouco armazenamento (menos que 50 gigabytes).

Além disso, *querys* específicas demoravam muito para serem processadas, então foram criados *index* e *views* no banco de dados para facilitar a leitura e consultas dos dados. Como consequência, devido aos indexes, o tamanho da base de dados ganha um pouco mais de volume, mas já contabilizados na informação anteriormente dada do espaço para armazenamento da base de dados.

9.2 OPORTUNIDADES

Apesar de todas as dificuldades, foi possível montar um *dataset* com diversas informações sobre vulnerabilidades e métricas de software associadas a artefatos afetados por elas. Esse conjunto de dados servirá como ponto de partida para novas investigações e replicações de pesquisas já realizadas.

Várias abordagens podem ser configuradas a partir dos dados dessa base, assim como foram utilizadas no Capítulo 5, só que podendo ser voltadas para artefatos menores como funções. As sugestões para investigação são: escolha dos projetos, validação cruzada, tipos de vulnerabilidades, balanceamento, seleção de atributos, algoritmos de aprendizagem de máquina e medidas de avaliação de resultado.

Para além disso, resultados podem combinar técnicas para aumentar a precisão e melhorar os resultados. É sabido que existem técnicas de detecção baseadas em padrões de códigos ou outras metodologias e elas podem ser tanto comparadas ou usadas combinadas para tentar melhorar as investigações na área.

Até algo mais além do que vulnerabilidades reportadas poderiam ser incorporado ao trabalho. Dados de vulnerabilidades detectadas pelas ferramentas e confirmadas por desenvolvedores poderiam ser incorporadas aos dados da base, o que garantiria mais instâncias vulneráveis e conhecimento melhor desse tipo de *bug* de software. Mas aí exige mais trabalho e muita validação de dados a ser realizada.

Este trabalho focou em vulnerabilidades reportadas, isso como já dito no Capítulo 7 é uma ameaça a validade do trabalho, mas essas instâncias são de vulnerabilidades genuínas que foram catalogadas por desenvolvedores, tornando assim uma categoria importante para ser estudada. Mais adiante trabalhos futuros podem melhorar esse quesito de coleta.

9.3 CONCLUSÕES

Este trabalho demonstrou a partir de dados estatísticos e observações manuais, a existência de relação entre métricas de software e vulnerabilidades de software. Entretanto a abordagem estatística deixa a desejar por deixar vago o nexo causal sobre essa relação, tendo resultado positivo mas não implicando em causalidade. Desta forma, também foi considerada a abordagem manual observativa, como já demonstrado na seção 2.3, com dez exemplos de vulnerabilidades afetando funções.

Para chegarmos a mais observações, era do nosso interesse investigar as melhores métricas para discriminar vulnerabilidades, então, para isso, foram separados métricas que se destacaram para realizar essa tarefa em três momentos. O primeiro momento foi as métricas citadas com grande variância e valores nos exemplos motivacionais, o segundo e terceiro momento foi na aplicação de algoritmos estatísticos para seleção das melhores *features*.

Por fim, chegou-se a resultados que destacam 6 métricas de software, são elas: *CountLine*, *CountPath*, *CountStmt*, *RatioCommentToCode*, *CountOutput* e *Knots*.

CountLine foi uma métrica que se destacou em todas as três abordagens, tando nas duas estatísticas quanto na manual. Claramente é fácil aceitar que quanto maior a quantidade de código, maior as chances de existir problemas e assim vulnerabilidades, uma reação natural. Entretanto, esse fato não é a causa da existência desses problemas, e sim consequência pelo aumento das possibilidades de exploração.

CountPath é uma métrica que contabiliza a quantidade de caminhos que um algoritmo pode realizar. Essa métrica se mostrou interessante em duas abordagens, a observação manual e a análise dos componentes principais. Essa métrica é também conhecida como McCabe's Cyclomatic, já foi bastante estudada e desde 1996 que o *Reliability Analysis Center* (RAC) (HARTZ; WALKER; MAHAR, 1996) recomenda que a métrica *McCabe*'s *cyclomatic complexity* com valor superior a 30 representa uma 'estrutura questionável' e maior que 50 o 'aplicativo não pode ser testado'. Isso se dá pelo fato de que fica inviável o desenvolvedor pensar em todas as possibilidades de funcionamento do algoritmo, então essa métrica no mínimo mostra indícios que o código tem problemas e comportamentos inesperados, podendo um desses caminhos ser um *ByPass* por exemplo, ou gerar um *Denial of Service*.

Outra métrica não muito diferente da fundamentação de *CountLine* é a *CountStmt*, só que ao invés de contar linhas, conta instruções. Uma linha de código pode esconder várias instruções, por isso acredita-se que, apesar de não ter apresentado forte correlação, essa seria uma métrica que colaboraria mais com a medida do crescimento do código. *CountLine* é usada com maior referência pois se trata de uma métrica mais visível às observações do desenvolvedor, entretanto não é uma métrica ideal para se medir o tamanho do código. *CountStmt* se destacou no PCA e nas observações manuais.

Pelas nossas observações, a métrica *RatioCommentToCode* serviu para discriminar

algumas das vulnerabilidades presentes nos 10 exemplos, entretanto isso reflete apenas uma convenção que seguem os desenvolvedores. Para os exemplos observados, foi percebido que os comentários eram mais alertas para o funcionamento ou a importância daquela determinada função ou trecho de código. Talvez isso seja apenas uma convenção da equipe de desenvolvimento dos projetos aqui escolhidos, alias nem todos apresentavam essa propriedade. O foco é mostrar que essas métricas irão variar de acordo com cada projeto, refletindo comportamentos e ações especificas de cada equipe que podem servir para fortalecer a segurança do sistema. Um exemplo hipotético, a métrica *RatioCommentToCode* foi boa para discriminar vulnerabilidades no projeto Derby porque era comum desenvolvedores alertarem em comentários sobre possíveis comportamentos e restrições que determinada função deveria seguir. Apesar de ser um exemplo hipotético, essa observação foi tirada do exemplo de vulnerabilidade do Tomcat e pela posição da métrica *RatioCommentToCode* que ficou em primeiro lugar com maior importância usando o algoritmo RFE, mas não é possível concluir tal argumentação pois não foram estudadas as convenções seguidas pela equipe de desenvolvimento desse projeto.

Uma métrica se destacou em todas as abordagens, foi a métrica *CountOutput*. Essa métrica conta a quantidade de chamadas de subprogramas ou variáveis globais pelo qual pode ser realizado alguma modificação de dados. Essa métrica está diretamente associada a acoplamentos da função e dados externos que ela modifica. Alterações de dados externos é uma atividade peculiar e pode trazer reações inesperadas, quanto maior o valor dessa métrica, mais improvável que o desenvolvedor saiba todas as reações que as modificações implicam. Ainda mais essas variáveis por serem externas, precisam ser validadas, e quanto mais a quantidade de entradas externas, mais robustez precisa ter a função para recebê-las.

Outra métrica se destacou com valores altos em certos casos foi *Knots*. *Knots* é uma métrica de grafo de controle de fluxo do algoritmo, entretanto ao invés de contar as rotas, ele mede a quantidade de nós no grafo. Simplificando, ele contabiliza os controladores de fluxo do algoritmo. O uso imprudente de controladores de fluxo pode gerar caminhos indesejados que resultem em funcionamentos inesperados e que podem comprometer a segurança do sistema. Essa métrica não é explicitamente visível ao desenvolvedor, mas pode mostrar situações assim como a métrica *CountPath* mostra.

Se observadas as qualidades que essas métricas descrevem, percebe-se que seus valores altos podem descrever que o código precisa de ajustes e que a complexidade do algoritmo saiu do controle da equipe de desenvolvimento. Mas, se olharmos por uma perspectiva profissional, programar é difícil e nem todos os problemas são solucionáveis com algoritmos simples. Se bem observado, é facilmente plausível que exista um nível de maturidade de complexidade para cada contexto de sistemas. A engenharia de software está crescendo para desenvolvimento de sistemas mais seguros. Percebe-se os avanços das tecnologias e paradigmas, tanto que é possível destacar que a preocupação em segurança

vem de longe e se fortificou com a chegada do paradigma de programação por serviços. Enquanto muitos módulos não se preocupavam com entradas e saídas externas porque esperavam que elas já fossem validadas, tal preocupação agora com SOA é notável, quando cada funcionalidade é tida como um serviço e é esperado que eles recebam todos os tipos de entradas, gerando uma certa preocupação com entrada e saída de dados em cada funcionalidade do sistema.

10 CONCLUSÃO

Neste trabalho, estudou-se a relação entre métricas de software e a existência de vulnerabilidades. Para isso, construiu-se um grande banco de dados com informações dos valores de métricas relacionadas a funções, classes e arquivos, e vulnerabilidades existentes nesses artefatos de código de sete projetos *open source*: Mozilla Firefox, Linux Kernel, Xen Hipervisor, Httpd, Glibc, Tomcat e Derby.

Neste trabalho, também foi realizado um estudo experimental avaliando 27 abordagens presentes no estado da arte de modelos de predição baseados em métricas de software para predição de vulnerabilidades. Essa avaliação foi realizada a partir dos dados do Tomcat e aplicada em cada abordagem com o mesmo conjunto de dados (alterando apenas as configurações e métricas escolhidas em cada abordagem). Os resultados ainda não são satisfatórios mas ficou claro novos desafios e problemas que atingem esse contexto de vulnerabilidades em artefatos de código. Eles serão demonstrados a seguir, em destaque as conclusões, descrição dos trabalhos futuros, os trabalhos em progresso e as considerações finais.

10.1 TRABALHOS FUTUROS

Claramente, vê-se a necessidade de segmentar ainda mais as vulnerabilidades pela sua diversidade, as vulnerabilidades de software não são problemas de códigos comuns e homogêneos, existem vários tipos de vulnerabilidades e essa especificação muitas vezes é determinada pela maneira como ela afeta o código (corrupção de memória). Logo, utilizar uma metodologia genérica para tentar identificar e correlaciona-las é um caminho obscuro e incerto, assim como mostrado nos resultados e análises deste trabalho.

Sendo assim, como trabalhos futuros, há a necessidade de estudar as vulnerabilidades segregando a partir de seus tipos ou contextos, dessa forma, é possível associar melhor as vulnerabilidades a certos tipos de métricas que podem caracterizar estilos de códigos específicos, visto que cada contexto de vulnerabilidade está associado a formas de código distintas.

É reforçada a ideia da necessidade de criação de novas métricas de software que discretizem melhor os artefatos, não adiantando ter inúmeras métricas de software se elas descreverem a mesma coisa. Como observado em algumas análises, há grupos de métricas que poderiam ser descritas por apenas uma ou pouco mais de duas métricas, que como processo de limpeza dos dados, vê-se necessário retirar essas redundâncias. Seguindo esse contexto, como trabalho futuro, é preciso estudar novos estilos de métricas de software, investindo até em métricas que vão além da complexidade estrutural.

Ainda várias outras investigações podem ser realizadas com as contribuições deste trabalho, é possível, por exemplo, usar o conjunto de dados para avaliar e comparar diversas configurações de modelos preditivos. O objetivo a longo prazo é construir modelos preditivos que possam identificar, em um projeto de software, quais são as funções com maior probabilidade de ter uma vulnerabilidade.

10.2 TRABALHOS EM PROGRESSO

Com o decorrer da pesquisa, percebe-se que existem várias possibilidades de aprendizagens de máquina e técnicas de pré-processamento de dados que podem ser aplicadas aos dados dessa base de dados. Mesmo com todas as configurações adotadas neste trabalho, ainda cada uma delas podem ser manipuladas com diversas configurações que podem melhorar os resultados de predição. Desta forma, fica disponível uma grande base de dados que pode ajudar investigadores na descoberta da relação entre complexidade estrutural e vulnerabilidades de software.

Desta forma, alguns trabalhos já estão sendo realizados fruto do produto e ideias deste. A base de dados foi requisitada para universidades como Universidade Federal de Pernambuco, Universidade de Coimbra, para a construção de modelos preditivos e Universidade Federal de Alagoas, para pesquisas em relação aos melhores algoritmos de aprendizagem de máquina para esse contexto.

10.3 CONSIDERAÇÕES FINAIS

A primeira consideração extraída com base nos resultados deste trabalho é que complexidade estrutural de código não é um fator único para se detectar e medir a propensão de artefatos de código terem vulnerabilidade. O fato da maioria das vulnerabilidades estarem presentes em funções mais complexas é relevante, mas diante de vários casos em que o contrário acontece, nos mostra que a questão exige mais investigação que apenas complexidade de código. Entretanto, os resultados das análises mostram que métricas de software podem ser utilizadas para discriminar código vulnerável e código neutro, mas supomos ser necessária a exigência de novas métricas e contextos diferentes de complexidade.

Algumas observações estão alinhadas com o esperado, mas também mostram algumas inesperadas como os resultados da seleção de *features* que aprovou a maioria das métricas. Os resultados também mostraram que as funções podem seguir uma propensão a conter novas vulnerabilidades, mas depende de projeto a projeto e isso se mostrou mais claro em projetos escritos na linguagem java. Foram encontradas correlações muito fortes entre métricas de software, exigindo a eliminação de redundâncias e exploração de novas métricas de software e finalmente, foi encontrada diferença estatística entre os grupos de métricas de funções vulnerável e métricas de funções neutras para a maior parte das métricas.

Por fim, foi possível observar que as métricas tradicionais como a precisão pode não ser muito útil nesses contextos, e também que a precisão e o *recall* podem ser ligeiramente enganosos, quando se utilizam conjuntos de dados desequilibrados. Métricas como *informedness* e *markedness*, por outro lado, são mais eficazes. Os resultados mostram que algumas das abordagens superaram outras, tendo melhores resultados com *decisions tree* e *random forest*, mas o que realmente alterou os resultados foram as escolhas das métricas.

REFERÊNCIAS

ABAEI, G.; SELAMAT, A. A survey on software fault detection based on different prediction approaches. **Vietnam Journal of Computer Science**, Springer, v. 1, n. 2, p. 79–95, 2014.

ABDI, H.; WILLIAMS, L. J. Principal component analysis. **Wiley interdisciplinary reviews: computational statistics**, Wiley Online Library, v. 2, n. 4, p. 433–459, 2010.

ALENEZI, M.; ABUNADI, I. Evaluating software metrics as predictors of software vulnerabilities. **International Journal of Security and Its Applications**, v. 9, n. 10, p. 231–240, 2015.

ALHAZMI, O.; MALAIYA, Y.; RAY, I. Measuring, analyzing and predicting security vulnerabilities in software systems. **Computers & Security**, Elsevier Advanced Technology Publications, v. 26, n. 3, p. 219–228, may 2007. ISSN 01674048.

ALVES, H.; FONSECA, B.; ANTUNES, N. Experimenting machine learning techniques to predict vulnerabilities. In: IEEE. **Dependable Computing (LADC), 2016 Seventh Latin-American Symposium on**. [S.l.], 2016. p. 151–156.

ALVES, H.; FONSECA, B.; ANTUNES, N. Software Metrics and Security Vulnerabilities: Dataset and Exploratory Study. In: **12th European Dependable Computing Conference (EDCC 2016)**. Gothenburg, Sweden: [s.n.], 2016.

ALVES, H. F.; ANTUNES, N.; FONSECA, B. **A Dataset of Source Code Metrics and Vulnerabilities**. 2016. Disponível em: <https://eden.dei.uc.pt/~nmsa/metrics-dataset>.

ANTUNES, N.; VIEIRA, M. On the Metrics for Benchmarking Vulnerability Detection Tools. In: **The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)**. Rio de Janeiro, Brazil: IEEE, 2015.

CATAL, C.; DIRI, B. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. **Information Sciences**, Elsevier, v. 179, n. 8, p. 1040–1058, 2009.

CATAL, C.; DIRI, B. A systematic review of software fault prediction studies. **Expert systems with applications**, Elsevier, v. 36, n. 4, p. 7346–7354, 2009.

CATAL, C.; SEVIM, U.; DIRI, B. Practical development of an eclipse-based software fault prediction tool using naive bayes algorithm. **Expert Systems with Applications**, Elsevier, v. 38, n. 3, p. 2347–2353, 2011.

CHOWDHURY, I.; ZULKERNINE, M. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In: **Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10**. New York, New York, USA: ACM Press, 2010. p. 1963. ISBN 9781605586397.

CHOWDHURY, I.; ZULKERNINE, M. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. **Journal of Systems Architecture**, Elsevier, v. 57, n. 3, p. 294–313, 2011.

CRAWLEY, M. J. **The R book**. [S.l.]: John Wiley & Sons, 2012.

DAVARI, M.; ZULKERNINE, M. Analysing vulnerability reproducibility for firefox browser. In: IEEE. **Privacy, Security and Trust (PST), 2016 14th Annual Conference on**. [S.l.], 2016. p. 674–681.

GOUSIOS, G.; ZAIDMAN, A. A dataset for pull request research. **Submitted to MSR**, 2014.

GRANITTO, P. M. et al. Recursive feature elimination with random forest for ptr-ms analysis of agroindustrial products. **Chemometrics and Intelligent Laboratory Systems**, Elsevier, v. 83, n. 2, p. 83–90, 2006.

HARTZ, M. A.; WALKER, E. L.; MAHAR, D. **Introduction to Software Reliability:State of the Art Review**. [S.l.]: Reliability Analysis Center, 1996.

HINKLE, D. E.; WIERSMA, W.; JURS, S. G. Applied statistics for the behavioral sciences. JSTOR, 2003.

HOVSEPYAN, A. et al. Software vulnerability prediction using text analysis techniques. In: ACM. **Proceedings of the 4th international workshop on Security measurements and metrics**. [S.l.], 2012. p. 7–10.

HOWARD, M. J. et al. Automatically mining software-based, semantically-similar words from comment-code mappings. In: IEEE PRESS. **Proceedings of the 10th Working Conference on Mining Software Repositories**. [S.l.], 2013. p. 377–386.

KAPLAN, J.; WEINBERG, A.; CHINN, D. Risk and responsibility in a hyperconnected world: Implications for enterprises. In: **World Economic Forum/McKinsey & Company**. [S.l.: s.n.], 2014.

LEWIS, J.; BAKER, S. The economic impact of cybercrime and cyber espionage. **Center for Strategic and International Studies, Washington, DC**, p. 103–117, 2013.

LOSSES, N. Estimating the global cost of cybercrime (2014), economic impact of cybercrime ii. **Center for Strategic and International Studies, McAfee/Intel Security**, 2014.

MCGRAW, G. **Software security: building security in**. [S.l.]: Addison-Wesley Professional, 2006.

MENEELY, A. et al. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In: ACM. **Proceedings of the 6th International Workshop on Social Software Engineering**. [S.l.], 2014. p. 37–44.

MENZIES, T.; GREENWALD, J.; FRANK, A. Data mining static code attributes to learn defect predictors. **IEEE transactions on software engineering**, IEEE, v. 33, n. 1, p. 2–13, 2007.

OHIRA, M. et al. A dataset of high impact bugs: Manually-classified issue reports. In: IEEE. **Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on**. [S.l.], 2015. p. 518–521.

OWASP, T. Top 10–2013. **The Ten Most Critical Web Application Security Risks**, 2013.

POWERS, D. M. Evaluation: from Precision, Recall and F-measure to ROC, Informedness, Markedness and Correlation. dez. 2011. ISSN 2229-3981.

RODRIGUES, D. et al. Engineering secure web services. **Crisis Management: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications**, IGI Global, p. 203, 2013.

SAMPAIO, L.; GARCIA, A. Exploring context-sensitive data flow analysis for early vulnerability detection. **Journal of Systems and Software**, Elsevier, v. 113, p. 337–361, 2016.

SCANDARIATO, R. et al. Predicting vulnerable software components via text mining. **Software Engineering, IEEE Transactions on**, IEEE, v. 40, n. 10, p. 993–1006, 2014.

SCITOOLS. **Understand™ Static Code Analysis Tool**. 2015. Disponível em: <https://scitools.com/>. Acesso em: 14 out. 2016.

SCITOOLS. **C++ Metrics**. 2016. <https://scitools.com/documents/metricImplementationNotes.pdf>. [Online; accessed 19-July-2016].

SHIN, Y. Exploring complexity metrics as indicators of software vulnerability. In: **Proceedings of the 3rd International Doctoral Symposium on Empirical Software Engineering, Kaiserslautem, Germany**. [S.l.: s.n.], 2008.

SHIN, Y. et al. On the use of calling structure information to improve fault prediction. **Empirical Software Engineering**, Springer, v. 17, n. 4, p. 390–423, 2012.

SHIN, Y. et al. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. **IEEE Transactions on Software Engineering**, v. 37, n. 6, p. 772–787, 2011. ISSN 0098-5589.

SHIN, Y.; WILLIAMS, L. An empirical model to predict security vulnerabilities using code complexity metrics. In: ACM. **Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement**. [S.l.], 2008. p. 315–317.

SHIN, Y.; WILLIAMS, L. Is complexity really the enemy of software security? In: ACM. **Proceedings of the 4th ACM workshop on Quality of protection**. [S.l.], 2008. p. 47–50.

SHIN, Y.; WILLIAMS, L. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In: ACM. **Proceedings of the 7th International Workshop on Software Engineering for Secure Systems**. [S.l.], 2011. p. 1–7.

SHIN, Y.; WILLIAMS, L. Can traditional fault prediction models be used for vulnerability prediction? **Empirical Software Engineering**, Springer, v. 18, n. 1, p. 25–59, 2013.

SMITH, B.; SHIN, Y.; WILLIAMS, L. Proposing sql statement coverage metrics. In: ACM. **Proceedings of the fourth international workshop on Software engineering for secure systems**. [S.l.], 2008. p. 49–56.

SOMMERVILLE, I. et al. **Engenharia de software**. [S.l.]: Addison Wesley São Paulo, 2003.

SPINELLIS, D. A repository with 44 years of unix evolution. In: IEEE PRESS. **Proceedings of the 12th Working Conference on Mining Software Repositories**. [S.l.], 2015. p. 462–465.

STUCKMAN, J.; WALDEN, J.; SCANDARIATO, R. The effect of dimensionality reduction on software vulnerability prediction models. **IEEE Transactions on Reliability**, IEEE, v. 66, n. 1, p. 17–37, 2017.

WALDEN, J.; STUCKMAN, J.; SCANDARIATO, R. Predicting Vulnerable Components: Software Metrics vs Text Mining. In: **2014 IEEE 25th International Symposium on Software Reliability Engineering**. [S.l.]: IEEE, 2014. p. 23–33. ISBN 978-1-4799-6033-0. ISSN 1071-9458.

WOHLIN, C. et al. **Experimentation in software engineering**. [S.l.]: Springer Science & Business Media, 2012.