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

Marcus Aurélio Cordeiro Piancó Júnior

Analisando as Relações entre Mudanças no Código Fonte e *bugs* no Software

Maceió - AL
2017

MARCUS AURELIO CORDEIRO PIANCO JUNIOR

Analisando as Relações entre Mudanças no Código Fonte e *bugs* no 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: Janis Christine Angelina Cavalcante

P581a Piancó Júnior, Marcus Aurélio Cordeiro.
Analisando as relações entre mudanças no código fonte e *bugs* no software / Marcus Aurélio Cordeiro Piancó Júnior. – 2018.
67 f. : il. grafs. tabs.

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

Bibliografia: f. 62-65.
Apêndices: f. 66-67.

1. Software. 3. Vulnerabilidade de software. 4. Bugs de software. 5. Código fonte - Mudanças. I. Título.

CDU: 004.05

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 Marcus Aurélio Cordeiro Piancó Júnior, intitulada: "*Analisando as Relações entre Mudanças e bugs no Software*", apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas em 19 de outubro de 2017, às 9h, na Sala "Alan Turing" do Bloco 12 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
Examinadora

"As convicções são inimigas
mais perigosas da verdade
do que as mentiras". Friedrich Nietzsche

AGRADECIMENTOS

Agradeço a Deus todo poderoso, que com sua infinita sabedoria me fez acreditar que nossos esforços estão sempre sob seus olhares, e que tudo na vida tem seu propósito e seu valor.

A ciência é fruto da necessidade do homem em melhorar a si e ao seu ambiente, e esta mesma ciência, que é capaz de transformar informações em curas de doenças, ou dados em realidades aumentadas, também tem o poder de transformar os indivíduos.

Assim, quero agradecer ás pessoas que, tal qual a ciência, que é transformadora, também foram para comigo. Primeiramente aos meus pais e a minha irmã, que durante todo esse percurso foram os pilares que sustentaram minhas lastimas e minhas alegrias, e me mostraram que deve-se ter paciência quando se deseja alcançar algo.

Quero agradecer também a minha Noiva e companheira Estefany que com sua paciência e sabedoria sempre me aconselhou para o bem, e me ajudou a traçar mais esse caminho.

Aos meus Orientadores, Baldoino Fonseca e Nuno Antunes que foram de uma postura ímpar durante todo o mestrado, e me ensinaram mais do que imaginam. Tenho por eles uma gratidão enorme, como também, agradeço aos meus professores Patrick, Evandro que sempre me ajudaram nos caminhos do aprendizado.

Quero agradecer também aos amigos que fizeram parte dessa jornada, em especial aos meus grandes Amigos Henrique Ferreira e Marcos José que muito me ajudaram durante o mestrado.Também agradeço aos meus amigos Wilk, Sebastião, João Lucas e outros que, direta ou indiretamente fizeram parte desta minha trajetória.

RESUMO

Com a evolução dos sistemas de software, alguns serviços, outrora realizados de maneira manual, têm migrado para aplicações computadorizadas, tais como: software complexas para controle financeiro ou sistemas que controlam vôos em aeronaves. Embora estes serviços tenham sido facilitados pelo auxílio da tecnologia, mantê-los disponíveis requer cuidados, seja durante a fase de desenvolvimento da aplicação que provê serviços ou pelos usuários que fazem uso destas aplicações. Um problema ainda predominante em sistemas é a ocorrência de *bugs* no software. Estes que, comumente, são associados a anomalias em unidades elementares dos sistemas, tais como arquivos, classes, funções ou métodos, podem ser demasiadamente longas ou complexas, o que pode comprometer as tarefas de inspeções e correções de *bugs*. Para tanto, este trabalho descreve o uso de um mecanismo de coleta e análise acerca das mudanças em códigos fonte que, comumente, podem estar associadas ao surgimento de *bugs* no software. Tal mecanismo baseia-se em modificações feitas ao longo do tempo em estruturas atômicas, e que são mudadas com frequência dentro do software, tais como: funções, métodos e classes. Utilizando-se da coleta e análise sobre dados de projetos, obteve-se um *data set* contendo o histórico de mudanças de quatorze projetos *open-source*, sendo três projetos desenvolvidos nas linguagens de programação *C* e *C++*, e onze projetos em *JAVA*. Foram realizadas também duas análises para avaliar o uso dos dados coletados, dos quais, sendo que a primeira foi uma análise acerca do histórico de mudanças associadas à vulnerabilidades de software (*bugs* de segurança) para os projetos C,C++, e a segunda acerca das relações entre mudanças e *bugs* em projetos *JAVA*. A partir das análises feitas, é possível afirmar que histórico de mudanças pode servir como referência a diversas estratégias que buscam o aprimoramento do processo de desenvolvimento de software, seja na criação de modelos para predição ou detecção de *bugs*, ou na geração de códigos de maneira automática.

Palavras-chave: *Bugs* de Software, Mudanças em Códigos Fonte, Histórico de Mudanças, Vulnerabilidade de Software.

ABSTRACT

Considering the evolution of software systems, some services that were performed manually have migrated to computer applications such as complex software for financial control or systems that control aircraft flights. Although these services have been facilitated by the help of the technology, keeping them available requires care, either during the development phase of the program that provides these services or by the users who make use of these application. A still prevalent problem in systems is the occurrence of *bugs* in the software. These are commonly associated with anomalies in elementary units in systems, such as files, classes, functions, or methods. These parts of source code can often be too long or complex, which can compromise the *bugs* inspections and fixes. In order to do so, this work describes the use of a collection and analysis mechanism, about the changes in source codes that, commonly, can be associated to the appearance of *bugs* in the software. Such a mechanism is based on modifications made over time in atomic structures, which are often changed within the software, such as functions, methods and classes. As a result of the collection and analysis engine applications, we obtained a data set containing information for the change history of fourteen open-source projects, three of which were developed in C and C ++ programming languages, and eleven projects in JAVA. Two analysis were also carried out to evaluate the use of the collected data, of which the first was an analysis of the history of changes associated with software vulnerabilities (security bugs) for C, C ++ projects, and the second about the relationships between changes and *bugs* in JAVA projects. From the analysis made, it is possible to affirm that history of changes can serve as reference to several strategies that seek to improve the process of software development, be it in the creation of models for prediction or detection of bugs, or in the generation automatically of source codes.

Keywords: Software Bug, Source Code Changes, Change History, Software Vulnerability.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Com a evolução dos sistemas de software, alguns serviços outrora realizados de maneira manual têm migrado para aplicações computadorizadas, tais como: softwares complexos para controle financeiro ou sistemas que controlam vôos em aeronaves. Embora estes serviços tenham sido facilitados pelo auxílio da tecnologia, mantê-los disponíveis requer cuidados, seja durante a fase de desenvolvimento do programa que provê serviços ou pelos usuários que fazem uso destas aplicações. É no contexto dos cuidados durante o processo de desenvolvimento de *software* que está incluído este trabalho.

Segundo (CLARKE; O'CONNOR, 2012) o desenvolvimento de software, em sua grande maioria, é feito de maneira integrada e de cooperação entre desenvolvedores. De modo que, sistemas desenvolvidos de maneira colaborativa tendem a possuir altos níveis de complexidade, o que torna o processo de manutenção do software uma tarefa difícil e em muitos casos, financeiramente inviável (CASEY; RICHARDSON, 2009).

É na fase de codificação do software, onde diferentes interesses entre os desenvolvedores podem ocasionar o aparecimento de *bugs* no sistema. Bugs estes que podem comprometer o funcionamento e a segurança do software. Para tanto, existem estratégias que tentam diminuir os impactos dos conflitos entre desenvolvedores, tais como: o uso de ferramentas para controle de versão do software (*GIT, SVN, Mercurial*) ou no uso de repositórios específicos para reportar problemas (*Bug reports*, *CVE-Details*) (CLARKE; O'CONNOR; LEAVY, 2016).

1.1 PROBLEMA

Um problema ainda predominante em sistemas é a ocorrência de *bugs* no software, que podem estar associados a exploração das unidades elementares do sistema, tais como componentes, classes, funções ou métodos. Estes trechos de códigos fonte, muitas vezes, podem ser demasiadamente longos ou complexos, o

que pode comprometer tarefas de inspeções e correções de *bugs*.

O processo de modificações, em funções ou métodos, está associado às mudanças feitas em códigos fonte. Tais mudanças geralmente ocorrem para fazer melhorias no software, sejam para a adição de uma nova funcionalidade, ou para a correção de *bugs*. Por outro lado, algumas mudanças mesmo que sejam para aprimorar ou evoluir algumas funcionalidades, podem, desencadear comportamentos anômalos, que são caracterizados por *bugs* no software.

Existem trabalhos na literatura que agrupam conjuntos de mudanças ocorridas em arquivos ou componentes de um software, para identificar possíveis anomalias. Tais agrupamento são denominados em alguns trabalhos de padrões de mudanças (DURAES; MADEIRA, 2006). Como os padrões de mudanças propostos por, (SOTO et al., 2016) para a correção de bugs, e os padrões de mudanças propostos por (DURAES; MADEIRA, 2006) para a identificação de *software faults*.

Embora existam na literatura, no âmbito da engenharia de software, especulações das relações de pertinência entre determinados padrões de mudança e a introdução de bugs em programas, ainda pouco se sabe sobre este fenômeno. Alguns autores na literatura simplesmente assumem ou especulam sobre a existência deste relacionamento (SHAR; TAN, 2012)(NEUHAUS et al., 2007)(BOSU et al., 2014). Porém, estes modelos não são baseados em evidências científicas convincentes que averiguem a relação de causa e efeito entre padrões de mudança e bugs em programas. Portanto, um problema prevalente na engenharia de software pode ser representado pela seguinte questão de pesquisa: **até que ponto certos padrões de mudanças em códigos fonte estão relacionados a bugs no software?**

Desenvolver estratégias para identificação de bugs de software, que possam fazer essa identificação com mais precisão, seja em níveis de funções, métodos ou classes, é bastante útil, do ponto de vista prático, nas tarefas de correções e inspeções de códigos fonte, pois o escopo de busca para corrigir *bugs* seria diminuído consideravelmente.

1.2 TRABALHOS EXISTENTES

Alguns trabalhos apresentam soluções que auxiliam o processo de desenvolvimento de software, ao mesmo tempo que propõem atenuar os efeitos de problemas no software como bugs no sistema. Dentre as soluções propostas, destacam-se as que detectam ou predizem bugs no software (NISTOR; JIANG; TAN, 2013)(GEGICK; ROTELLA; XIE, 2010)(KIM et al., 2006)(KIM; JR; ZHANG, 2008). Já outras soluções analisam mudanças feitas durante as correções de bugs para determinar prováveis padrões e sugerir de maneira automática, correções futuras (HERZIG; ZELLER, 2013)(SOTO et al., 2016)(PAN; KIM; WHITEHEAD, 2009).

Existem também abordagens que fazem uso de dados extraídos de códigos fonte (PALOMBA et al., 2013)(FLURI et al., 2007), e que apresentam modelos modelos de detecção de bugs no software, usando uma estratégia específica denominada de *change extractor* para coletar estes dados com um nível maior de granularidade.

Em particular, no trabalho desenvolvido por (SHIN et al., 2011) são considerados os números de alterações em arquivos para identificar bugs de segurança. O estudo utiliza métricas denominadas de *code churn*. Estas métricas são compostas pelos números totais de modificações em arquivos, tais como: número de linhas adicionadas, número de linhas removidas e número de linhas modificadas.

(SHIN et al., 2011) investiga padrões de mudanças considerando a temporalidade em que esses padrões ocorrem. De maneira geral, para cada commit (modificação) em um determinado arquivo (reportado com *bug* ou sem *bug*), são coletados valores absolutos das métricas de *code churn* (mencionadas no parágrafo anterior), que juntas determinam padrões que , segundo os autores, são capazes de distinguir arquivos vulneráveis de não vulneráveis.

De uma perspectiva prática, analisar arquivos, a depender do tamanho do software, pode se tornar uma tarefa dispendiosa. Também deve-se levar em conta que existem taxas consideráveis de falsos alarmes. Portanto, ainda que segundo (SHIN et al., 2011) existem fortes indícios de uma relação estreita entre, padrões de mudança e bugs de segurança em arquivos, estes arquivos podem ser demasiadamente longos, comprometendo assim, o processo de inspeção e correção de *bugs*. Caso as métricas de *code churn* fossem associadas a tipos específicos de modificações, padrões mais

específicos poderiam ser criados, tornando possível a localização de *bugs* sobretrechos de códigos menores, tal como métodos ou funções. Dessa forma, a tarefa de inspeção ou correção de bugs no código seria uma tarefa menos dispendiosa.

Já (NEUHAUS et al., 2007) analisa o histórico de alterações em componentes que tenham sido reportados com bugs de segurança. Mais especificamente, esta abordagem propõe um mapeamento em componentes de um software que ainda utilizam chamadas de funções ou *imports*, de funcionalidades advindas de componentes do sistema com bugs de segurança, identificando assim trechos que ainda usam estes componentes e possam conter *bugs*.

Um problema prevalente na abordagem proposta por (NEUHAUS et al., 2007), se dá no processo de descoberta dos componentes com *bugs* de segurança, pois só são mapeados caso já existam *reports*. Com isso, apenas produtos de software que tenham apresentado correções de bugs de segurança são capazes de utilizar a abordagem proposta. Além disso, componentes podem conter, uma grande quantidade exorbitante de arquivos, o que inviabilizaria uma inspeção mais detalhada acerca dos possíveis bugs de software.

Já na abordagem descrita por (BOSU et al., 2014),os bugs são identificados utilizando uma estratégia de análise das revisões feitas em trechos de códigos fonte. A abordagem também considera os diferentes perfis dos desenvolvedores envolvidos com as modificações. (BOSU et al., 2014) tenta combinar informações sobre aspectos estruturais do código e dos desenvolvedores para identificar possíveis bugs de software. De modo que, os diferentes perfis dos desenvolvedores foram agrupados por tempo de desenvolvimento em projetos, e quantidade de commits que foram revisados e submetidos na trilha principal dos repositórios dos projetos.

Um desafio no processo de criação de modelos, sejam para predição ou detecção de *bugs*, é a diminuição da taxa de ruídos durante a fase da extração dos dados. Para os estudos que analisam mudanças em códigos fonte deve-se considerar que algumas modificações podem não estar associadas a ocorrência, ou a correção de *bugs* (SOTO et al., 2016).Estima-se que, cerca de 16.6% dos artefatos (códigos fonte) modificados nas correções de *bugs*, não estejam associados com modificações corretivas (HERZIG; ZELLER, 2013), o que deixa uma considerável margem para erros

em modelos que propõem detectar ou prever *bugs* baseados em mudanças em códigos fonte.

1.3 OBJETIVO DA PESQUISA

1.3.1 Objetivo Geral

O objetivo deste trabalho é analisar as relações entre tipos específicos de mudanças em códigos fonte e *bugs* de software.

1.3.2 Objetivos Específicos

Para atingir o objetivo geral supracitado, os seguintes objetivos foram considerados:

- Entender mudanças realizadas em códigos fonte;
- Avaliar metodologias de extração de mudanças já existentes na literatura;
- Desenvolver uma metodologia de coleta das mudanças em códigos fonte;
- Implementar mecanismos para coleta dos dados;
- Extrair dados de projetos para diferentes domínios e linguagens de programação;
- Estabelecer uma representação dos dados coletados;
- Definir uma classificação para as mudanças;
- Realizar análises estatísticas para averiguar as relações entre mudanças e bugs.

1.4 ORGANIZAÇÃO DO TEXTO

O restante desta dissertação está estruturado da seguinte forma:

- **Capítulo 2 – Fundamentação Teórica**, contém conceitos fundamentais para a compreensão do texto.

- **Capítulo 3 – Trabalhos Relacionados**, neste capítulo são descritos alguns estudos que desenvolvem mecanismos de coleta para o histórico de mudanças, tendo como foco algum tipo de análise específica sobre esse histórico.
- **Capitulo 4 – Processo para Coleta dos Dados**, aqui é descrito o processo de coleta do histórico de mudanças proposto neste trabalho.
- **Capítulo 5 – Análise da Relação entre Mudanças e Bugs**, capítulo onde são feitas análises sobre os dados de mudanças em códigos fonte e *bugs* no software
- **Capítulo 6 – Conclusão**, neste capítulo é feita uma síntese dos resultados obtidos nas análises, bem como futuras análises que podem ser feitas.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são introduzidos conceitos fundamentais para o entendimento do trabalho.

2.1 DESENVOLVIMENTO DE SOFTWARE

Desenvolver software de maneira colaborativa demanda, em muitos casos, alta complexidade na tarefa de manter um sistema consistente em muitas versões do software. Tal complexidade se dá mediante a falta de sincronia no processo de manutenção ou criação de uma funcionalidade no sistema, uma vez que desenvolvedores podem modificar trechos que ainda estão sendo escritos por outros desenvolvedores, isso pode causar conflitos de interesses ou até possíveis problemas no sistema, tais como *bugs* de software.

Existe uma solução bastante difundida, que são os sistemas de controle para versionamento, que surgiu em meados dos anos 80 com a proposta de auxiliar os desenvolvedores na tarefa de evolução e versionamento dos softwares (TICHY, 1985). Os sistemas para controle de versão evoluíram, podendo ser utilizado no desenvolvimento colaborativo de grandes projetos, por desenvolvedores de diversas partes do mundo. Um dos sistemas para controle de versão mais usados atualmente é o *git*, desenvolvido no ano de 2005 por (HAMANO; TORVALDS, 2005), para ser utilizado no desenvolvimento do *linux kernel*. Muitas equipes de desenvolvimento de código aberto usam o *git*, não apenas para compartilhar códigos, mas também para partilhar soluções de problemas, e outras informações relevantes aos projetos.

Fazer uso de sistemas para controle de versão traz consigo alguns desafios quando se quer analisar grande quantidade de dados. Por exemplo, o número de *commits* (submissões de alterações no projeto) pode chegar na casa das centenas de milhares, tornando o uso de algumas estratégias de inspeção sobre códigos fonte uma tarefa difícil. Todavia, a análise das modificações feitas ao longo do tempo, seja em trechos de códigos fonte, ou nos reports (*logs*) de problemas no software, podem

evidenciar padrões que estejam associados a bugs no sistema.

2.2 HISTÓRICO DE MUDANÇAS EM CÓDIGOS FONTE

Como mencionado na seção anterior, o processo de desenvolvimento pode ser feito de maneira colaborativa, utilizando ferramentas para auxiliar no controle de versões, e nas soluções de problemas e conflitos entre os desenvolvedores.

Uma das grandes vantagens das ferramentas de controle para versões, é o armazenamento das mudanças feitas ao longo do tempo no software, denominado em muitos estudos como *change history*. Tal histórico é amplamente utilizado para revisões em códigos, ferramentas de geração de *code snnipets*, e no desenvolvimento de estratégias para detecção ou predição de bugs no software, como nos estudos de (COUTO et al., 2014) (PALOMBA et al., 2013)(BOSU et al., 2014).

Alguns estudos mencionados no parágrafo anterior fazem uso de diferentes estratégias para mineração de dados em repositórios VCS (Version Control System), no processo de extração de dados sobre as mudanças. Outros trabalhos fazem uso de *parsers*, em conjunto com expressões regulares, para identificar padrões de mudanças, enquanto outros usam uma estratégia mais sofisticada, que minimiza a taxa de erros sobre a extração de mudanças, desenvolvendo *scripts* de scaneamento sobre ASTs (*Abstract Sintaxe Tree*). ASTs são representações em árvore da estrutura sintática abstrata do código-fonte escrita em uma determinada linguagem de programação, onde os tipos de mudanças são explicitados como nós da *tree*.

Estratégias que utilizam mineração sobre *ASTs* para coletar o histórico de mudanças tendem a errar menos, pois os dados são extraídos diretamente na *AST*, que é criada para ser validada pelo analisador sintático da linguagem de programação. Já estratégias que usam padrões sobre expressões regulares, para capturar mudanças em códigos fonte, tendem a ter uma alta taxa de defeitos, pois as diversas variações na escrita em linguagens de programação, podem conter variações não encontradas pelos padrões definidos nas expressões regulares.

2.3 *ORTHOGONAL DEFECT CLASSIFICATION* COMO CLASSIFICADOR DAS MUDANÇAS

Diversas mudanças são realizadas durante o ciclo de vida de um software, tornando necessário o uso de mecanismos que possam agregar um significado a tais mudanças a fim de melhor representá-las. Para tanto, utilizou-se uma metodologia de classificação denominada *Orthogonal Defect Classification* (ODC) (CHILLAREGE et al., 1992).

As justificativas para utilizar uma adaptação da classificação ODC, para representar mudanças de software, são: i) os padrões de mudança utilizados para a classificação de defeitos têm um significado implícito e são similares aos que pretendemos analisar; ii) a metodologia ODC tem um forte suporte teórico que fornece alguma semântica às mudanças realizadas em níveis de código fonte. Tais fatores são relevantes para o cumprimento dos objetivos deste trabalho, uma vez que espera-se averiguar as relações entre, possíveis padrões de mudanças e bugs no software.

As categorias definidas pela *ODC* são capazes de cobrir grande parte das mudanças de códigos fonte, segue a definição da classificação na *ODC* segundo (DURAES; MADEIRA, 2003):

- *Assignment*: valores atribuídos incorretamente ou não atribuídos;
- *Checking*: validação ausente ou incorreta de dados ou loop incorreto ou declarações condicionais;
- *Interface*: erros na interação entre componentes, módulos, *drivers* de dispositivos, declarações de chamadas, ou listas de parâmetros;
- *Timing/serialization*: falta ou serialização incorreta de recursos compartilhados;
- *Algorithm*: implementação incorreta ou faltante que pode ser corrigida implementando um algoritmo ou estrutura de dados sem a necessidade de uma mudança de design;
- *Function*: refere-se a trechos de códigos que estão implementados incorretamente ou não implementados.

Além dos classificadores citados acima, na *ODC* existem três categorias de mudanças: i) *Missing construct*, que consiste em mudanças que foram adicionadas, e estavam faltando em algum trecho do código fonte; ii) *Wrong construct*, que define as mudanças que tiveram uma construção incorreta, ou não condizente com seu propósito; iii) *Extraneous construct*, que representa mudanças que foram removidas, por não serem usadas, ou por já estarem em desuso.

2.4 *BUGS* DE SOFTWARE

Bug é uma condição acidental que faz com que uma unidade funcional não execute a função necessária no sistema (ELECTRICAL; ENGINEERS, 1989). Descobrindo os *bugs* após o software ser colocado em modo de produção, o custo finaceiro e de manutenção podem aumentar consideravelmente, o que pode acarretar em significativos prejuízos financeiros. Entender e identificar fatores que levaram à incidência de um *bug* é uma tarefa difícil e demorada, visto que ele pode surgir em diferentes partes do sistema, podendo afetar todos ou alguns componentes de uma só vez.

2.4.1 Tipos de Bugs

Existem diversas definições para tipos diferentes de *bugs*. Os exemplos que seguem abaixo são descritos no estudo feito por (OHIRA et al., 2015), que inicialmente são suficientes para determinar o sub-conjunto de instâncias com *bugs* utilizadas no experimento e análise feita na Seção 5.

1) *Bugs* de segurança: um bug de segurança (GEGICK; ROTELLA; XIE, 2010) pode gerar um problema sério que muitas vezes afeta as utilizações de produtos de software diretamente. Uma vez que os dispositivos de Internet (por exemplo, smartphones) e seus usuários estão aumentando a cada ano, os problemas de segurança de produtos de software devem ser de interesse para muitas pessoas. Em geral, os estas falhas de segurança devem ser corrigidos o mais rápido possível. Por se tratar de um tipo de *bug* com impactos negativos, muitas vezes, incalculáveis, esse tipo específico de bug foi explorado com mais detalhes neste trabalho.

2) *Bugs* de desempenho: um bug de desempenho (NISTOR; JIANG; TAN, 2013) é definido como "erros de programação que causam degradação significativa do desempenho". A "degradação do desempenho"é decorrente de pouca experiência do usuário, capacidade de resposta da aplicação preguiçosa, menor rendimento do sistema e desperdício de recursos computacionais (MOLYNEAUX, 2009). (NISTOR; JIANG; TAN, 2013) mostraram que um *bug* de desempenho precisa de mais tempo para ser corrigido que outros tipos de *bugs*. Portanto, os erros de desempenho podem afetar os usuários por um longo tempo.

3)*Bugs* de Robustez: um *bug* de robustez (SHIHAB et al., 2011) é "um *bug* associado a funcionalidade do sistema, que é comumente introduzido em trechos de código fonte modificados para adicionar novos recursos ou para corrigir *bugs* já existentes". Este tipo de *bug* pode tornar versões utilizáveis de um sistema em versões inutilizáveis.

2.5 VULNERABILIDADE DE SOFTWARE

Bugs de segurança, podem ser especificados também como sendo vulnerabilidades de software. Tal Vulnerabilidade é responsável por grande parte dos investimentos nas atividades de prevenção e correção de falhas de segurança. Estima-se que mais de $100 bilhões de dólares são anualmente investidos em segurança cibernética, em particular nas atividades de prevenção ou manutenção de produtos de software. Segundo prognósticos recentes (LEWIS; BAKER, 2013), esses números tendem a crescer.

Em muitos casos, estas vulnerabilidades podem permitir que atacantes violem as políticas de privacidade, ou permissões de um sistema, e através destas, explorem o software para fins maliciosos (JIMENEZ; MAMMAR; CAVALLI, 2009). Muitos destes *bugs* de segurança causadores de vulnerabilidades no software, são advindos do processo de implementação ou manutenção do software (ZEGEYE; SAILIO, 2015). Considerando a relevância deste tipo de *bug* produziu-se neste trabalho uma análise específica norteada pelas mudanças de códigos reportados como vulneráveis na seção 5.1.

2.5.1 Exemplos de vulnerabilidades de software

Alguns tipos de vulnerabilidades de software têm comportamentos distintos e podem ocorrer de diversas maneiras e em cenários distintos. Portanto, investigar cada vulnerabilidade levando em consideração seus diferentes tipos, pode ser relevante no processo de melhoria das práticas de codificação de um software, como também para entender qual a melhor maneira de corrigir estas vulnerabilidades de software.

Seguem alguns exemplos de vulnerabilidades de software, que segundo o estudo de (CHRISTEY, 2007) foram reportadas com maior frequência, e têm um alto grau de relevância para diversos aspectos da qualidade de um software.

- *Buffer overflow*: Geralmente, ocorre quando um *array* ou vetor, que armazena uma determinada quantidade de dados pré-definida é submetido a tentativas de armazenar uma maior quantidade de dados do que o tamanho já estabelecido. Isso possibilita ao atacante executar comandos que possam alterar o comportamento do sistema (JIMENEZ; MAMMAR; CAVALLI, 2009).

- *Integer Overflows*: Esta vulnerabilidade ocorre de duas maneiras. Em um dos casos a vulnerabilidade é explorada quando, diante de operações de conversão de tipos "Inteiros", por exemplo, quando *signed integer* é convertido para um *unsigned integer* e a variável que armazena os dados em *signed integer*, não suporta o formato convertido. O segundo caso ocorre quando, operações aritméticas resultam em um "inteiro longo", e a variável que receberá o valor resultante da operação aritmética, comporta apenas dados do tipo "inteiro comum". Tanto no primeiro, como no segundo caso, a vulnerabilidade ocorre quando as estruturas não comportam tipos “Inteiros” de dados maiores dos que podem armazenar. Com essa vulnerabilidade o atacante pode prejudicar o funcionamento (JIMENEZ; MAMMAR; CAVALLI, 2009).

- XSS(*Cross-site scripting*): Consiste em uma vulnerabilidade geralmente de aplicações web, em que o atacante insere trechos de código em paginas *WEB*, no lado do cliente, onde são executados fragmentos de códigos *JavaScript*, que têm a finalidade de executar rotinas ou comandos, que possam dar acesso ao atacante ou causar danos ao usuário (JIMENEZ; MAMMAR; CAVALLI, 2009).

- *SQL Injection*: Consiste na inserção de trechos de código, trechos estes que podem acessar o banco de dados, e manipular informações. Normalmente, ocorre quando códigos que são sensíveis ao contexto de um banco de dados, são executados tendo como alvos variáveis ou campos de textos, que representem entradas de dados de usuários, ou de solicitações feitas por outras funções do sistema. Comumente, tal vulnerabilidade surge por falta de validações, corretas, nos trechos responsáveis pela entrada de dados no sistema (JIMENEZ; MAMMAR; CAVALLI, 2009).

3 TRABALHOS RELACIONADOS

Este capítulo descreve alguns estudos que utilizam as mudanças efetuadas em códigos fonte, como recursos para gerir modelos de predição, ou de detecção para problemas no software.

Adotar mecanismos que facilitem o processo de inspeções em códigos fonte não é uma tarefa trivial. A dificuldade em gerir ferramentas que facilitem a visualização sobre trechos de códigos fonte modificados ao longo do tempo, durante o processo de desenvolvimento de software, ainda é um desafio para a engenharia de software. Alguns projetos são demasiados extensos, e possuem grandes quantidades de LOC (Lines Of Code) por arquivo. (YOON; MYERS; KOO, 2013) propõe uma solução de auxílio aos desenvolvedores no processo de visualização das mudanças em códigos fonte ao longo do tempo, permitindo que o desenvolvedor inspecione apenas o que é realmente de interesse no momento desejado.

Nesse contexto, (YOON; MYERS; KOO, 2013) desenvolveram um plugin denominado de *AZURITE*, que consiste em um editor de comandos no *eclipse*, que realiza consultas que retornam visualizações de mudanças feitas no software. O plugin dispõe de duas abordagens de visualização: 1) visualização em forma de *time line* das mudanças, permite ao desenvolvedor navegar nas modificações em códigos de maneira fácil, focando no que se deseja; e um uma funcionalidade denominada de *code history diff view*, permitindo que o desenvolvedor navegue pelas modificações em um trecho específico de código fonte, e avalie todo seu histórico.

Uma das limitações desse trabalho (YOON; MYERS; KOO, 2013) é a incapacidade de analisar projetos fora do escopo da plataforma *eclipse*, pois os *plugins* produzidos sobre esta plataforma tem alta dependência com componentes do próprio *eclipse*. Além disso, o *plugin* pode não funcionar em versões diferentes do *eclipse*, caso alguma funcionalidade na plataforma mude ou deixe de existir. Outra desvantagem desse trabalho se encontra na análise de grandes projetos, como por exemplo o projeto *linux kernel* ou *mozilla*, que detêm de mais de meio milhão de submissões de códigos fonte, de maneira que a visualização provida pelo *plugin* seria demasiadamente

complexa.

Diversos são os problemas advindos da não adoção de boas práticas durante a fase de desenvolvimento de software. Alguns destes problemas têm impactos diretos na qualidade do software, como por exemplo, a inserção de *bad smells* no código fonte, que afetam, dentre outras coisas, os processos de inspeção ou reuso do código. Na tentativa de atenuar os impactos causados por *bad smells*, (PALOMBA et al., 2013) desenvolveram uma abordagem que detecta cinco diferentes tipos de *code smells*. Denominada de *HIST* (*Historical Information for Smell deTection*), para projetos *JAVA*, que consiste na análise do histórico de mudanças em códigos fonte, para desenvolver modelos de detecção de *bad smells*.

A abordagem *HIST* seguiu um processo de análise sobre *code smells* que se diferencia da grande maioria dos estudos, estes usam, geralmente, análises estática para detectar *smells*. Além disso, (PALOMBA et al., 2013) mostraram que o uso do histórico de mudanças para detectar estruturas que evoluem para possíveis *bad smells*, pode ser mais útil do que algumas estratégias existentes, uma vez que é preferível que o desenvolvedor adote medidas corretivas para possíveis *code smells* durante a fase de desenvolvimento, a correções com o software já em funcionamento. Uma das limitações da abordagem proposta por (PALOMBA et al., 2013), é a necessidade de se ter um histórico de mudanças considerável para que a ferramenta tenha bons resultados, essa limitação é prevalente na maioria das estratégias que fazem uso do histórico de mudanças como fonte de análise.

Além dos problemas que afetam a qualidade do software, existe uma categoria de *bugs* que tem causado prejuízos bilionários às empresas e organizações, que são os *bugs* de segurança.

A fim de evitar que *bugs* de segurança no software tenham maiores impactos sobre os sistemas, empresas e pesquisadores têm demonstrado interesse em prever ou detectar a ocorrência de *bugs* de software (JIMENEZ; MAMMAR; CAVALLI, 2009). Algumas das estratégias propostas pela literatura acerca de predição ou detecção têm como objetivo diminuir o escopo de inspeções necessárias, e que possam indicar com alguma acurácia, partes do software propensas a conter algum bug de segurança.

Entretanto, inspecionar partes de um software propensas a conter possíveis

bugs de segurança tem se mostrado uma tarefa difícil e cara. É difícil pois, em muitos casos, a inspeção de trechos ou partes que detêm *bugs*, pois necessitam do conhecimento prévio acerca do comportamento e do tipo de bug de segurança. É custosa pois, a depender do tamanho do sistema, as inspeções podem crescer de maneira a impossibilitar a inspeção de todas as partes do software propensas a conter possíveis *bugs* de software (JIMENEZ; MAMMAR; CAVALLI, 2009).

No estudo feito por (SHIN et al., 2011), foram desenvolvidos modelos para predição de *bugs* de segurança (Vulnerabilidades de *sofwtare*). Com efeito, os modelos neste trabalho foram criados a partir de um conjunto de métricas bem específicas:

1. Métricas associadas a complexidade do software;
2. Métricas que avaliam os diferentes perfis dos desenvolvedores, e qual a influência deste perfil na introdução de possíveis *bugs* de segurança;
3. Métricas denominadas de *code churn* (ou métricas de mudanças), onde são extraídas as quantidades totais de modificações em um arquivo para cada *commit* modificador, ao longo do tempo.

Para conjunto de métricas denominado de *code churn*, foram definidos três características para definir as mudanças em arquivos, são elas: i) o número total de linhas modificadas em um arquivo, ii) o número total de linhas adicionadas em cada arquivo, e iii) o total linhas em que apenas foram feitas mudanças.(SHIN et al., 2011) realizaram uma análise considerando mudanças em níveis de arquivo. Analisar mudanças em arquivos pode representar um risco ao processo de concepção de modelos de predição, pois não distingui-se durante o estudo tipos de mudanças, apenas valores absolutos com os totais de modificações foram associados às métricas de *code churn*. Mudanças em trechos de códigos associados às funcionalidades complexas foram igualmente contabilizados, da mesma maneira que as modificações feitas em comentários de códigos ou documentação.

Os projetos analisados por (SHIN et al., 2011) foram, *Mozilla Firefox web browser*, e o *Red Hat Enterprise Linux kernel 4*. Para o primeiro foram coletados dados de 34 versões de desenvolvimento ao longo de quatro anos, contendo mais de 10.000 arquivos e mais de dois milhões de linhas de código, e para o segundo *Red Hat*

Enterprise Linux kernel 4 foram mais de 13.000 arquivos analisados e cerca de três milhões de linhas de código.

Os autores afirmam que, 24 das 28 métricas investigadas conseguem discriminar arquivos vulneráveis de arquivos não vulneráveis. Além disso, segundo os autores, com a combinação dos conjuntos de métricas definidos no trabalho é possível prever vulnerabilidades (*bugs* de segurança) no software com uma precisão de 80% (SHIN et al., 2011).

Para o sub-conjunto de métricas de mudanças definidas por (SHIN et al., 2011), deve-se levar em consideração as seguintes observações durante a criação dos modelos de predição:

- Geralmente, os arquivos que foram considerados vulneráveis eram os que detinham de grande quantidade de linhas de código, o que torna uma possível correção ou inspeção algo difícil e custoso. Uma provável solução seria analisar partes menores, tais como métodos ou classes, indicando quais desses trechos poderiam ser vulneráveis, diminuindo assim o escopo de inspeçãa de códigos;
- Os projetos analisados, ainda que tenham sido projetos de grandes proporções, e de grande relevância para a comunidade, estão limitados a dois domínios, sendo um *web browser* e outro um componente *core* para sistemas operacionais. Seriam então necessários outros experimentos, para que os modelos de predição pudessem se mostrar mais abrangentes;

Já no trabalho proposto por (BOSU et al., 2014), analisou-se o histórico de *code reviews*, onde foram reportadas possíveis *bugs* de segurança no software. Além disso, foram identificadas quais características de mudanças de código são propensas a conter estes *bugs*. (BOSU et al., 2014) também considerou a experiência dos desenvolvedores como fator relevante para a inserção de *bugs* de segurança. No estudo foram analisados 267,046 *code review* para 10 *source projects*, foram identificadas 413 *bugs* de segurança associados aos códigos que poderiam ser explorados por possíveis atacantes. Os resultados de (BOSU et al., 2014) sugerem que podem existir relações de pertinência entre mudanças em códigos fonte, a experiência do desenvolvedor e *bugs* de software.

(BOSU et al., 2014) também confirmaram que as modificações em códigos fonte vulneráveis são comumente escritos por desenvolvedores mais experientes, o que comprova o estudo feito por (MCGRAW, 2004) onde é descrito que independente da experiência, em sua grande maioria, os desenvolvedores não tem conhecimento de práticas de codificação seguras. Já os desenvolvedores com menos experiência também tendem a inserir *bugs*, porém em um número muito maior, entre 1.5 até 24 vezes mais do que desenvolvedores com experiência.

No trabalho proposto por (HOVSEPYAN et al., 2012) foi desenvolvida uma abordagem de análise sobre códigos fonte, utilizando técnicas de mineração de texto no processo de criação de modelos para identificação de *bugs* de segurança de software. Foram analisadas versões do *K9 email client* da plataforma *Android*. A técnica de aprendizagem de máquina utilizanda no processo de criação dos modelos foi a *Supervisor Vector Machines* (SVM). Os resultados, ainda que preliminares, indicam uma acurácia de 87%, com uma precisão de 85% e um *recall* de 88%. O processo de criação dos modelos se deu mediante a classificação de vetores produzidos a partir de monogramas (trechos de códigos fonte) onde foi estabelecido um conjunto de artefatos que comporiam estes vetores, tais como: classes, modificadores de acesso, tipos de retorno, chamadas de função e etc, tendo sido originados a partir do total de ocorrência de cada artefato. Como mostrado na Figura 1, que gerou o vetor *"class:1, HelloWorldApp:1, public:1, static:1, void:1, main:1, String:1, args:1, System:1, out:1, println:1"*, de maneira que, o par de cada elemento do vetor representa o tipo de estrutura de código fonte, e a quantidade de ocorrências desta estrutura respectivamente.

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

Figura 1 – Exemplo de Código Gerador do Vetor no estudo de (HOVSEPYAN et al., 2012)

Embora essas técnicas apresentem resultados promissores do ponto de vista

do desenvolvimento de modelos para predição ou detecção de *bugs*, as abordagens descritas nos parágrafos anteriores ainda apresentam limitações que devem ser levadas em consideração, tais como:

- São capazes de identificar somente estruturas de alto nível, tais como, arquivos ou componentes de um software propensos ao surgimento de *bugs* de segurança. Embora esta identificação seja importante, é muito comum que algumas destas estruturas sejam demasiadamente extensas. Portanto, não é muito útil para os desenvolvedores conhecerem quais estruturas (arquivos ou componentes) estão relacionadas com o surgimento de possíveis *bugs*, uma vez que os desenvolvedores necessitam realizar inspeções com um maior nível de detalhe. Seria mais útil para um desenvolvedor indicar quais partes (funções ou métodos), possuem maior probabilidade de surgimento de possíveis *bugs* de software.
- Foram avaliadas apenas em pequenos conjuntos de dados, como nos estudos feitos por (SHIN et al., 2011), onde apenas algumas versões do *Mozilla* e uma versão do *Linux Kernel* foram avaliados, e na abordagem proposta por (NEUHAUS et al., 2007) onde foram analisados apenas *bugs* de segurança do *Mozilla* para um pequeno conjunto de versões. Como consequência, os modelos propostos pelas duas abordagens podem ser comprometidos, pois o comportamento dos *bugs* de segurança podem ter sofrido alterações, tanto em versões anteriores como em versões posteriores, podendo assim comprometer os resultados dos modelos de identificação. Isso torna difícil verificar a aplicação destes modelos até mesmo em versões diferentes das versões investigadas, pois os modelos de identificação podem não se comportar como o esperado em diferentes contextos dos avaliados pelos estudos feitos.

Além dos trabalhos que têm como principal foco a detecção ou predição de *bugs* no software, existem também estudos que se preocupam nos processos de correções destes *bugs*, algumas técnicas fazem uso também do histórico de correções de *bugs*, para verificar a existência de padrões determinantes, que são frequentes nas correções (SOTO et al., 2016) (PAN; KIM; WHITEHEAD, 2009). Os padrões de correção de *bugs* em ambos os estudos são especificamente extraídos de projetos *JAVA*.

Tanto (SOTO et al., 2016), quanto (PAN; KIM; WHITEHEAD, 2009) definem os padrões de correção de *bugs* como sendo a combinação dos tipos específicos de códigos fonte definidos na gramática formal da linguagem de programação *JAVA*, com o tipo de modificação feita, tais como, adições, remoções ou alteração, de trechos de código.

No estudo realizado por (SOTO et al., 2016), além dos padrões de correções formulados, foi levado em conta quais tipos de estruturas existiam no fragmento de código antes da correção dos *bugs*, e por quais tipos foram substituídos durante a correção dos *bugs*, estabelecendo assim, uma relação entre os tipos que existiam antes, e por quais foram substituídos depois da correção dos *bugs*. Um exemplo dessa substituição é um fragmento de código fonte *JAVA* contendo o tipo *IFStatement* alterado por um tipo *SwitchCase*. Segundo os autores, ao estabelecer essa relação de *replacing* entre estruturas, é possível melhorar a eficácia das ferramentas de reparo automático para *bugs*, visto que, é possível chegar a menor granularidade no código fonte.

Um dos problemas relevantes que deve ser levado em consideração no trabalho desenvolvido por (SOTO et al., 2016), está no processo de extração das informações sobre as correções dos *bugs*. Nesse procsso, os dados coletados são advindos de uma ferramenta denominada *BOA*, que tem como principal funcionalidade a execução de *queries* sobre projetos em repositórios *git*, sem considerar prováveis erros dentro desta ferramenta. Outro problema é a homogeneização dos domínios para os projetos analisados, pois, projetos de domínios distintos podem conter comportamentos distintos, o que influenciaria na maneira como são tratadas as correções dos *bugs*.

Já no estudo de (PAN; KIM; WHITEHEAD, 2009), os padrões de mudanças para correção de *bugs*, foram definidos usando a mesma estratégia de (SOTO et al., 2016), combinando os tipos de estruturas provenientes da gramática formal da linguagem de programação *JAVA*, com o tipo de modificação feita, tais como, adições, remoções ou modificações, de trechos de código. Como no caso do padrão *"Method Call with Different Number of Parameters or Different Types of Parameters"*, de sigla *MC-DNP*, em que se representa parte do fragmento de código correspondente a correção de um *bug*, e é determinado pela mudança dos parâmetros dentro da chamada de método, como mostrado abaixo:

- query = getLuceneQuery(filter.getFilterRule());
+ query = getLuceneQuery(filter.getFilterRule(), analyzer);"

No total, (PAN; KIM; WHITEHEAD, 2009) definem 27 padrões recorrentes em correções de *bugs*, que segundo os autores, conseguem cobrir entre 45.7% a 63.6% de todas as soluções aplicadas aos *bugs* reportados pelos projetos analisados. Destes 27, dois tiveram uma melhor representatividade, nas resoluções dos *bugs*, que foram os padrões associados a mudanças em condicionais (19.7%–33.9%) e chamadas de métodos (entre 21.9%–33.1%).

Um problema similar nas duas abordagens de (PAN; KIM; WHITEHEAD, 2009) e (SOTO et al., 2016), é no processo de coleta das informações acerca das correções dos *bugs* (arquivos e *commits*). Pois, foram considerados reports que em muitos casos não eram *bugs*, e foram reportados como sendo. Outro problema nas duas abordagens é a não avaliação da taxa de mudanças que não faziam parte das correções e podem influenciar a criação dos modelos para correção automática de *bugs*.

4 PROCESSO PARA COLETA DOS DADOS

O processo evolutivo de um software, seja para a adição de novas *features* ou para adequar um programa a um comportamento novo, exige mudanças em códigos fonte. Mudanças estas que são introduzidas, na maioria dos casos, de maneira incremental, e por mais de um desenvolvedor. Muitos trabalhos fazem uso dessas mudanças para criar estratégias que tentam aprimorar o desenvolvimento de software, tornando-o mais seguro e com menos problemas, como visto na seção anterior. Para tanto, o processo de análise sobre mudanças de códigos fonte requer que os registros das mudanças sejam extraídos com a menor taxa de erro possível.

Extrair dados sobre mudanças em códigos fonte não é uma tarefa trivial. Diversos são os motivos pelos quais esta tarefa se torna dispendiosa e difícil. Dentre eles se destacam:

1. O tamanho dos projetos que podem chegar a casa das milhões de linhas de códigos modificadas, tornando a extração de informações complexa e demorada;

2. Alguns projetos são escritos sem que sejam seguidos os protocolos para reportar problemas ou modificações conflitantes;

3. A diversidade de linguagens de programação, muitas vezes, obriga que estratégias de extração de mudanças sejam restritos a um contexto específico e limitado pela linguagem;

4. Altos custos computacionais envolvidos. O processo de extração de mudanças pode demorar meses a depender do tamanho do projeto, comprometendo assim os recursos envolvidos.

Para avaliar se existem relações de pertinência entre mudanças e *bugs*, inicialmente, é preciso coletar estas mudanças. Por isso, este trabalho propõe um mecanismo de extração das modificações feitas no software, seja nas mudanças feitas ao longo do tempo, ou em algum *snapshot* específico. Todavia, no estágio atual deste mecanismo

as coletas consideradas são: i) sobre o histórico de mudanças em arquivos; ii) em *snapshots* de arquivos envolvidos com problemas no software, tais como: bugs e vulnerabilidades de software. Nas duas coletas é possível determinar quais as classes, métodos ou funções envolvidas na mudança, o que proporciona um menor nível de granuralidade que pode ser explorado durante as análises.

Muitas são as vantagens ao se analisar o mudanças no software, tais como, desenvolvimento de modelos para predição ou detecção de *code smells*, vulnerabilidades no software, bugs, como mencionado na Seção de 3.

As principais etapas no processo de coleta de mudanças são:

1. **Extração das Mudanças Envolvidas nas Correções dos *Bugs***: É o processo de extração dos elementos (arquivo,*commit*,*bug_id*) envolvidos durante as correções dos *bugs* no software. A partir destes elementos é possível chegar a um nível atômico das mudanças, identificando quais as classes, métodos ou funções foram mudadas para corrigir o *bug*. Além disso, é possível identificar os tipos específicos de mudanças envolvidas nas correções, tais como: condicionais, *loops*, variáveis, invocação de métodos. Um resumo deste processo pode ser visto no quadrante superior da Figura 2. Este processo pode ser feito de duas maneiras, coletando os elementos na *web reports* de *bugs*, tais como: *bugzilla*, *cve-details*, *Jira Bug Tracking*. Outra maneira de coletar estes elementos é através do repositório *git* do projeto, que em muitos casos, tem *reports* das correções de *bugs* bem definidos.

2. **Extração do Histórico de Mudanças**: É o processo que coleta todas as modificações feitas ao longo do tempo em um arquivo, seja para os arquivos com *report* de *bugs*, ou em arquivos sem este *report*. Dessa maneira, é possível usar os históricos dos arquivos com e sem *reports* de bugs para realizar análises comparativas, ou criar modelos baseados em distinção entre amostras. O ultimo quadrante da Figura 2 resume esse processo.

É importante observar que nos processos de extração das mudanças, para arquivos que foram reportados com *bugs* no software, a coleta correta dos elementos de correção (arquivo, *commit*,*bug_id*) é de suma importância. Além disso, requer o

conhecimento mínimo acerca de como funciona o processo de uso de um sistema de controle de versões (por exemplo *git*).

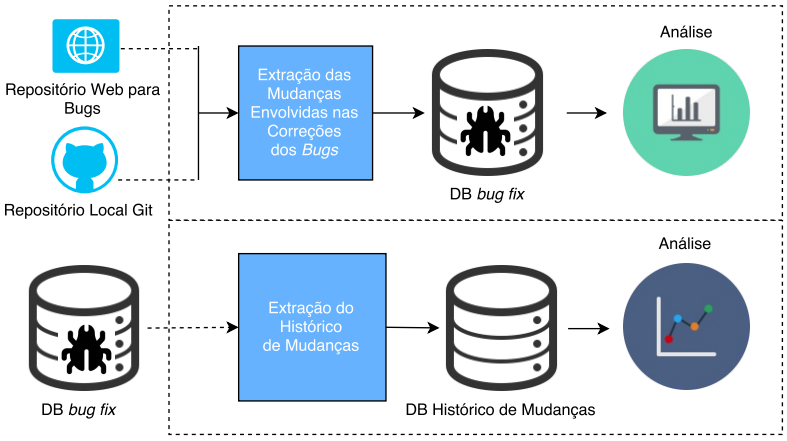

Figura 2 – Fluxograma da Coleta das Mudanças

Além disso, devem ser considerados dois pré-requisitos na escolha do projeto do qual as mudanças serão extraídas:

- O projeto deve possuir um sistema para controle de versões (para esta abordagem especificamente o *git*). Todavia, caso o projeto utilize outras ferramentas para controle de versões do sistemas, é possível adaptar alguns comandos para que a coleta dos elementos envolvidos com *bugs* possam ser extraídas corretamente.
- O projeto deve possuir um mecanismo de *report* para bugs disciplinado, sejam estes mantidos no repositório (*git*) local do projeto,como visto na Figura 3, ou em *webreports*, como mostra na Figura 4,

```
commit 1d86c9c3d128f64afa51460e3ed7900963755cfd
Author: Arjen Poutsma <apoutsma@pivotal.io>
Date:   Thu Jul 27 16:08:29 2017 +0200

    Use Credentials object instead of 2 attributes for Basic Authentication

    This commit changes the usage of two separate attributes (username and
    password) into one: a single `Credentials` object.
    Additionally, the attributes key under which the credentials are stored
    is changed to be specific to Basic Authentication, in order to allow for
    other sorts of authentication later.

    Issue: SPR-15764
```

Figura 3 – Exemplo de um *bug/issue* report no repositório local

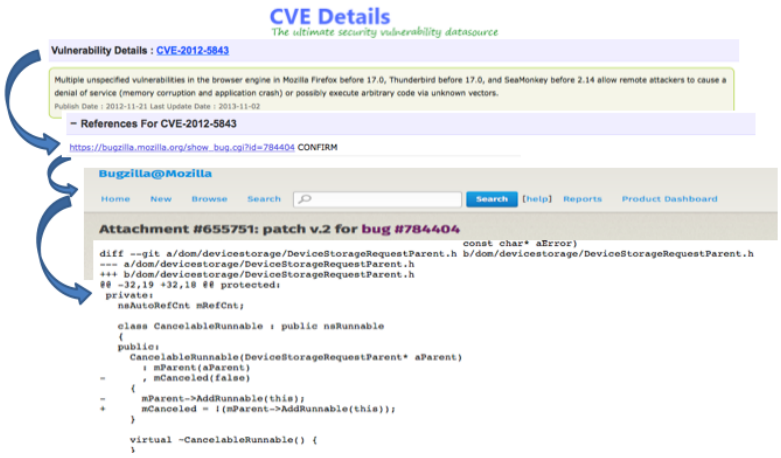

Figura 4 – Exemplo do report para um *bug* de segurança na *web*

Com o objetivo de avaliar a metodologia de coleta das mudanças descrita nos parágrafos anteriores, foi desenvolvida duas ferramentas de coleta, sendo a primeira para extração dos elementos associados com as correções dos *bugs* no software descrita na seção 4.1; e a segunda um extrator de mudanças ao longo do tempo, descrita na seção 4.2. A ferramenta foi desenvolvida utilizando a linguagem de programação JAVA, e com o auxílio de algumas *APIs*, tais como: *CDT*, *JDT*.

A ferramenta pode ser baixada no endereço <https://github.com/MarcusPianco/changeExtractor>, onde são descritos alguns exemplos simples, bem como outras informações úteis.

4.1 EXTRAÇÃO DAS ESTRUTURAS ASSOCIADAS A *BUGS* NO SOFTWARE

Como mencionado anteriormente, a extração das estruturas envolvidas nas correções dos bugs, podem ser feitas usando duas estratégias de coletas diferentes, a depender da disponibilidade das informações acerca das informações dos bugs para cada projeto. Essas informações podem estar na *web*, ou podem estar dispostos através do repositório do projeto. Um resumo dos processos pode ser visto nas Figuras 5 e 6.

A extração dos elementos usados nas correções dos *bugs* no repositório do projeto, é feito em três etapas:

1. Após a escolha do projeto, dadas as especificações já mencionadas anteriormente, é preciso fazer o *download* do repositório do projeto;

2. Na segunda etapa é necessário indicar o caminho local do repositório no campo correspondente na ferramenta de extração, bem como indicar qual a *keyword* que representa um *report* de *bug,issue* (derby-3532), ou no caso de *bugs* de segurança(cve-2017-14746);

3. Na última etapa é preciso observar os exemplos de saída mostrados na ferramenta. Caso exista alguma divergência ou erro, deve-se observar melhor o uso da *keyword*. keyword. Persistindo os erros pode ser necessário alterar a ferramenta para as adequações necessárias. Nesta etapa a ferramenta disponibiliza comandos específicos para validação das mudanças extraídas.

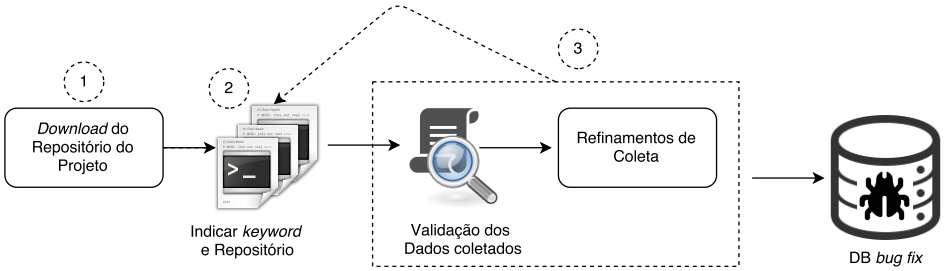

Figura 5 – Representação do processo de coleta dos elementos no repositório do projeto.

Na coleta feita nos *web reports* o processo de extração é feito a partir de buscas nas páginas *htmls* dos *bugs report*, ou através de uma *api rest* disponibilizada pela equipe mantenedora do projeto. Esse processo é demorado e tem uma taxa de erro na coleta, ainda que pequena, quando comparada à coleta no repositório do projeto.

Quando não há a disponibilidade de uma *api rest* no *bug report*, a ferramenta coleta os elementos envolvidos nas correções com o auxílio de uma *api* denominada de *Jsoup*, que tem como principal função inspecionar *tags* em uma página *html* e

retornar conteúdos a partir de seletores específicos. Essas duas maneiras de coletar são mostradas na Figura 6

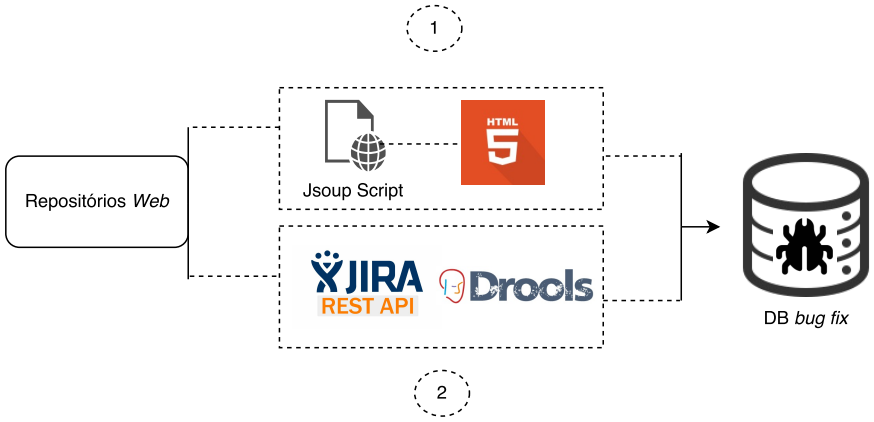

Figura 6 – Representação do processo de coleta dos elementos na *web*

Nas duas abordagens, a coleta dos elementos modificados para as correções dos bugs são sempre os mesmos, um fragmento de informação contendo uma tupla "*commit*_correção, arquivo_corrigido, *bug_id*".

4.2 EXTRAÇÃO DO HISTÓRICO DE MUDANÇAS

Especificamente para este trabalho, a ferramenta desenvolvida para a coleta de mudanças está vinculada à coleta das mudanças sobre arquivos com *reports* de *bugs*. Todavia, ferramenta de coleta do histórico de mudanças pode ser usada em outros contextos, desde que o projeto siga as mesmas recomendações citadas anteriormente.

Dessa maneira, após serem coletados os arquivos, e os *commits* envolvidos nas correções dos bugs, o próximo passo é extrair quais mudanças foram feitas ao longo do tempo nestes arquivos. Nesta fase também são determinadas quais estruturas (classes, métodos ou funções) foram modificadas dentro do arquivo.

O ponto de partida para a coleta do histórico de mudanças, se dá a partir do segundo *commit* feito no arquivo associado com as correções dos bugs. Considera-se que o commit de criação, ainda que possa ter sido a submissão geradora de possíveis

bugs no *software*, não é por si um *commit* de correção de *bugs*, podendo desencadear muitos *outliers* nos dados coletados.

O processo para extração do histórico de mudanças é ilustrado na Figura 7. Onde as siglas, "CI"representa o *commit* inicial, "CF"como sendo o *commit* final, e "CM"como sendo o commit modificador entre o intervalo do commit inicial e o final (*commit*), que pertencem ao conjunto de todos os *commits* modificadores de um arquivo. A Figura 7 está representando diferentes históricos de mudanças, onde os círculos em vermelho, vistos no primeiro fluxo de modificações, denotam a persistência do *bug* em um arquivo, e o circulo azul representa o *commit* de correção do bug. Já o segundo fluxo representa as mudanças feitas em um arquivo, onde a ocorrência de bugs ainda não foi reportada.

A extração do histórico de mudanças nos arquivos sem reports de bugs, só é necessária quando se quer realizar análises comparativas para identificar discrepâncias, se existirem. Particularmente neste trabalho estes dados foram coletados, e posteriormente usados como fonte de dados a serem comparados.

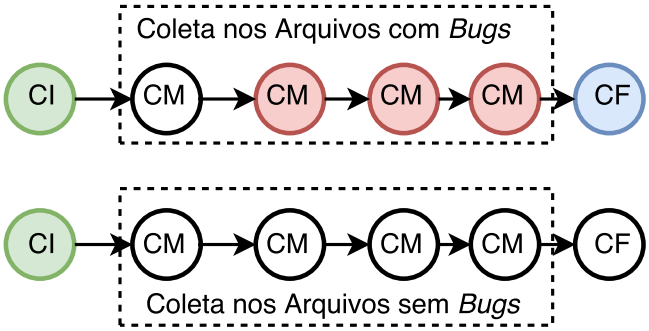

Figura 7 – Representação do processo de Extração do Histórico de Mudanças

A coleta do histórico de mudanças também é representada pela notação intervalar na equação: {(ci,cf)={cm $\varepsilon\{C/\}ci < cm < cf\}$}.

Onde *"ci"*,*"cf"* representam o *commit* inicial de coleta e *"cf"* o *commit* final, respectivamente, *"cm"* os *commits* modificadores, e *"C"* representa o conjunto de todos os *commit* de modificações em um arquivo.

É possível alterar os intervalos de extração do histórico e mudanças, podendo estabelecer períodos específicos e de acordo com a necessidade de cada análise. Neste trabalho, foram feitas duas análises, descritos nas Seções 5.1 e 5.2 , nas quais os intervalos para a coleta do histórico de mudanças são bem específicos.

A estratégia utilizada para extrair as mudanças se dá mediante a coleta dos conjuntos dos números de linhas alteradas no arquivo pelo *commit* modificador. Na Figura 8, os quadrantes verticais representam o conteúdo de uma alteração em um arquivo por um *commit*, e o número exato da linha alterada pelo *commit* no arquivo, respectivamente.

Neste processo a base de dados que é gerada contém informações representadas por uma tupla, onde em cada coleta das mudanças dadas pelo par *commit*/arquivo contêm os seguintes dados: *commit*, arquivo, conjunto dos números de linhas modificadas e o identificador do bug, sendo que este último dado é opcional, a depender se o arquivo teve ou não *report* de *bug*. Essa base de dados serve como entrada para o mecanismo de classificação das mudanças, descrito na Seção 4.3. O Algoritmo 1 que descreve os processos descritos neste parágrafo, é mostrado a seguir.

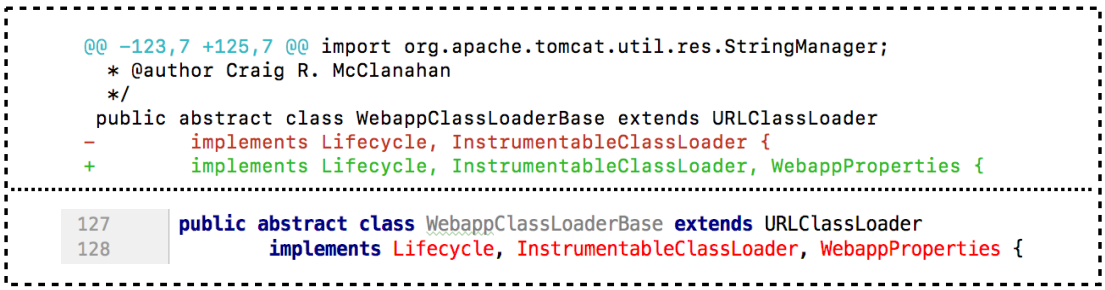

Figura 8 – Localização das Linhas Modificadas No Arquivo.

4.3 CLASSIFICAÇÃO DAS MUDANÇAS

Depois de coletar o histórico de mudanças, é necessário classificar e atribuir sentido às mudanças de acordo com o que se pretende avaliar. Por exemplo, as métricas da ODC (Orthogonal Defect Classification) descritas na Seção 2.3, foram usadas para dar semântica às modificações na primeira análise deste trabalho. Já nas

Algorithm 1 Algoritmo para Coleta do Histórico de Mudanças

```
1: procedure CHANGEEXTRACTOR(commit_fix, file_fix)
2:     conjuntolinhasmodificadas[]
3:     lista_commits[] = intervalodecommits(file_fix, commit_fix)
4:     for each commit i in lista_commits do
5:         linhasmodificadas = coletarlinhasmudadas(i)
6:         conjuntolinhasmodificadas.add(linhasmodificadas)
7:     end for
8:     Return conjuntolinhasmodificadas
9: end procedure
```

outras duas análises as mudanças foram associadas a tipos específicos de estruturas contidas na linguagem de programação JAVA, de modo que o uso da ODC também pode ser adaptada a essas modificações facilmente.

Inicialmente, decidiu-se realizar o processo de classificação utilizando reconhecimento de padrões através do uso de expressões regulares, criando padrões que pudessem reconhecer cada estrutura mudada e atribuir um tipo de classificação específica. Entretanto, alguns tipos de mudanças seguem padrões de implementação que variam de linguagem para linguagem. Verificou-se então, que algumas modificações não estavam sendo catalogadas. Fez-se necessário utilizar uma abordagem mais sólida e independente de padrões.

Então, criou-se uma abordagem de classificação baseada em *parsers* de localização de tipos na *AST* (Árvore de Abstração Sintática). Usou-se duas *APIs* CDT (BINH VIET HOANG ANH, 2009), para os projetos escritos em C,C++, e JDT, para os escritos em *Java*. Foram realizadas múltiplas buscas, tomando como referência conjuntos de números de linhas modificadas da base de dados, para determinar o tipo específico de cada nó na AST.

Com o uso desta classificação, ou atribuição de tipos às mudanças é possível realizar trabalhos com um nível maior de granularidade, visto que, analisar trechos menores, como métodos ou funções, para produzir modelos de detecção ou predição de bugs, diminui o escopo de inspeção dos desenvolvedores na tarefa de correção dos bugs no software.

4.4 COMPOSIÇÃO DOS DADOS

Nesta seção são apresentados os dados coletados pelo mecanismo de extração de mudanças.

A dificuldade em obter dados acerca dos bugs em projetos open-source, é algo relatado com frequência em alguns estudos (KIM; JR; ZHANG, 2008) (OHIRA et al., 2015) (KIM et al., 2006). Seja pela ausência de *report* dos desenvolvedores, ou por não possuir critérios disciplinados que façam os desenvolvedores reportarem as correções, segundo (KIM; JR; ZHANG, 2008), apenas cerca de 40%-60% dos *bugs* são reportados nos repositórios de *bug tracking*[1]

Os dados provenientes da extração das mudanças podem facilitar os estudos que tentam determinar se existem diferenças significativas, e que possam indicar, através de testes estatísticos, se mudanças em códigos fonte podem ser representativas na diferenciação entre arquivos que possam gerir problemas dentro do software. Segue alguns exemplos de análises, que podem ser feitas através das informações advindas do histórico de mudanças:

- Agrupamento das mudanças mais frequentes em estruturas reportadas com prováveis bugs;
- Agrupamento das modificações feitas no processo de correção dos bugs no software, na tentativa de encontrar algum padrão ou tendência;
- Avaliar até que ponto o histórico de mudanças, agrupado por tipos específicos de mudanças, pode diferenciar estruturas (classes, métodos, funções, etc) que possam evoluir de maneira a gerir bugs no software;
- Determinar se existe similaridade ou dissimilaridade entre históricos de mudanças em arquivos com e sem *bugs*.

Além destes quatro exemplos, existem vários outros que podem ser feitos através da análise sobre mudanças em códigos fonte, como os estudos descritos

[1] Repositório, geralmente, mantido na *web*, que provê mecanismos de *report* para *bugs*, onde as soluções destes são discutidas e analisadas por mais de um desenvolvedor. Comumente são reportados *commits* e arquivos que foram manipulados para a correção de *bugs*.

na Seção 3. Todavia, a abordagem desenvolvida neste trabalho, se diferencia das demais. Em muitos estudos referentes a relação entre o histórico de mudanças, não é descrito se durante o processo de extração do histórico das mudanças em arquivos, as referências aos arquivos movidos de um pacote para outro são mantidas em um único histórico de mudanças.

Conforme a estratégia de coleta do histórico de mudanças é possível indexar arquivos que mudaram de pacotes, fazendo com que a análise e os dados sobre o histórico de mudanças sejam mais confiáveis. Isso pode ajudar na diminuição da taxa de erros em técnicas que utilizam tal histórico como entrada de dados para modelos de detecção, ou predição de bugs.

4.5 DETALHAMENTO DOS DADOS COLETADOS

Tanto nos projetos escritos na linguagem de programação *C,C++* ou nos projetos *JAVA*, a representação dos dados coletados, seja para a coleta das estruturas associadas às correções dos bugs ou para o histórico de mudanças, é mantida a homogeneidade na persistência dos dados. A relação dos dados se dá como mostrado na Figura 9, em que o identificador para bugs da tabela *bug_fix_structures* indexa a tabela *change_history* e só vai existir se arquivo tiver *report* de *bugs*.

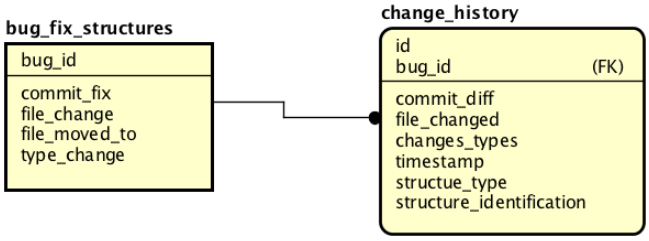

Figura 9 – Diagrama de Entidade Relacionamento entre o histórico de mudanças e as estruturas modificadas durante a correção dos bugs.

Como resultado dos dados coletados a partir da metodologia descrita na Seção 4.1, obteve-se os dados acerca das estruturas relacionadas às correções dos *bugs*, para quatorze projetos *open-source*. A escolha dos projetos seguiu os critérios estabelecidos

no Capitulo 4. Dos quatorze projetos, onze são escritos na linguagem de programação *JAVA* e três são escritos nas linguagens *C,C++*. As Tabelas 1 e 2 sintetizam os dados coletados durante a extração das mudanças associadas às correções dos *bugs*. Destes, apenas três tiveram o histórico de mudanças coletados, pois o tempo de coleta das modificações é demasiadamente demorado. É importante observar que na quarta coluna algumas quantidades de arquivos *Java* associados à correções são maiores que a quantidade total de arquivos *Java*, isso se dá pois são considerados também os arquivos já removidos, e que foram modificados no passado em correções de *bugs*.

Tabela 1 – Projetos *Java* que tiveram as coletas das estruturas associadas às correções de *bugs* feitas.

Projetos	Arquivos Totais	Arquivos Java	Arquivos Java Associados as Correções de Bugs/Issues	Bugs/Issues Reported	Java LOC *
Camel	20635	17086	17714	7704	1062813
Derby	3388	2845	3355	3203	691384
Tomcat	2827	2245	1870	1757	306667
Commons-Lang	352	318	872	655	75422
Spring Framework	7535	6724	10472	4693	582962
Teiid	2727	2254	6772	2732	363825
Hibernate-orm	10120	9051	9535	3418	657254
Hibernate-search	2456	2375	2576	1379	155604
Hibernate-validator	1780	1627	1952	743	81081
Wicket	4711	3218	5116	3504	213268
WildFly	9961	8021	9533	1736	479820

Tabela 2 – Projetos *C,C++* que tiveram as coletas das estruturas associadas às correções de *bugs* feitas.

Projetos	#correções	# total de funções	# Func. Vulneráveis
Mozilla	567	60947	505
Kernel	443	34653	1279
Total	**1010**	**95600**	**1784**

Para os projetos *C,C++*, a coleta do histórico de mudanças, bem como das estruturas envolvidas com as correções dos bugs, foi feita apenas em dois projetos o *Linux Kernel* e o *Mozilla*. Essa escolha se deu tanto pela relevância dos projetos, quanto pela facilidade inicial em coletar os dados. Para estes projetos também foram consideradas estruturas específicas (funções e métodos) na coleta do histórico de mudanças e nas correções dos bugs. Já nos projetos *JAVA*, apenas o projeto *tomcat*, teve o histórico de mudanças obtido. As bases de dados, tanto para o histórico de mudanças, quanto para as mudanças responsáveis pelas correções dos *bugs*,

são totalmente independentes da metodologia aqui proposta. Com isso, outras abordagens podem utilizar tais dados em outros estudos.

Com o objetivo de entender melhor as relações entre mudanças e *bugs*, foram feitas duas análises que são descritas na seção seguinte.

5 ANÁLISE DA RELAÇÃO ENTRE MUDANÇAS E *BUGS*

Neste Capítulo são descritas as análises acerca das relações entre Mudanças e bugs no Software.

5.1 AS RELAÇÕES ENTRE MUDANÇAS E *BUGS* DE SEGURANÇA

Esta primeira análise tem como principal objetivo identificar as relações de pertinência entre mudanças em códigos fonte e vulnerabilidades de software (bugs de segurança).[1]

Com a evolução dos sistemas de software, diversos serviços críticos, tais como, transações bancárias e compras de produtos, são usados, de forma ágil e frequente. Embora tais serviços tenham proporcionado vários benefícios, seu uso requer cuidados, bem como seu desenvolvimento.

Em particular, várias práticas devem ser adotadas para garantir a segurança dos sistemas, evitando que bugs de segurança estejam presentes no software. Caso contrário, é possível que algum atacante ou software mal intencionado possa explorar tais vulnerabilidades para benefício próprio ou para causar prejuízos financeiros a companhias e organizações.

Nesta análise, avaliamos o histórico de mudanças em funções e métodos, para identificar com o menor nível de granularidade, a existência de diferenças significativas, que possam distinguir modificações ao longo do tempo, entre funções reportadas com *bugs* de segurança, e funções que não tiveram estes *reports*. Foram definidas oito métricas para representar as mudanças feitas nas funções analisadas. Os resultados, ainda que preliminares, mostraram que das oito métricas de mudança,

[1] Para todos os efeitos, durante esta análise, *bugs* de segurança e vulnerabilidades de software, são termos que definem a mesma categoria de problemas no software. Pois, os reports de vulnerabilidades inicialmente são reportados como bugs de segurança.

apresentadas na Tabela 3, seis foram capazes de distinguir, funções vulneráveis (com bugs de segurança), de funções não vulneráveis.

Para entender cada mudança coletada, foram realizadas adaptações sobre a classificação das mudanças descritas na *ODC*. Assim, foram definidas as métricas que representam as modificações no software. As métricas que representaram as mudanças já classificadas nesta primeira análise, podem ser vistas na Tabela 3.

Tabela 3 – Classificadores usados, adaptadas para ODC (CHILLAREGE et al., 1992).

	Extraneous	Missing	
	NEC	NMC	Conditional Statement
ODC Types	NEFC	NMFC	Function Call
	NEM	NMM	Method
	NEV	NMV	Variable

As métricas apresentadas na Tabela 3 fazem parte de dois grupos da *ODC*: (**Extraneous**) que representa um trecho de código removido pelo desenvolvedor; e (**Missing**) que representa trechos de códigos que estavam faltando e foram adicionados pelo desenvolvedor, como visto na seção 2.3. Segue abaixo a especificação de cada métrica usada na representação das mudanças.

***Number Extraneous Conditional* - NEC:** Número de Condicionais que foram removidos no processo de mudança;

***Number Missing Conditional* - NMC:** Número de Condicionais que foram adicionados no processo de mudança;

***Number Extraneous Function Call* - NEFC:** Número de Chamadas de Funções que foram removidos no processo de mudança;

***Number Missing Function Call* - NMFC:** Número de Chamadas de Funções que foram adicionadas no processo de mudança;

Number Extraneous Method - NEM: Número de Funções ou Métodos que foram removidos no processo de mudança;

***Number Missing Method* - NMM:** Número de Funções ou Métodos que foram adicionados no processo de mudança;

***Number Extraneous Variable* - NEV:** Número de declarações ou atribuições á variável que foram removidos no processo de mudança;

***Number Missing Variable* - NMV:** Número de declarações ou atribuições á variável que foram adicionadas no processo de mudança;

Embora outras métricas da *ODC* pudessem ser consideradas durante esta análise, investigou-se apenas um subconjunto destes classificadores ODC. A escolha das métricas tomou como base o estudo de (DURAES; MADEIRA, 2006), em que essas métricas foram consideradas como fortes indicadores a conter falhas. Sendo assim, nesta análise será investigado a aptidão destas métricas para representar trechos de código com bugs de segurança.

5.1.1 Análise Estatística

Para analisar se existe uma diferença estatisticamente significativa entre os valores das métricas relacionadas a funções vulneráveis e não vulneráveis, aplicou-se alguns testes estatísticos (WOHLIN et al., 2012).

Para cada métrica, os testes foram usados para comparar duas amostras diferentes: uma com instâncias vulneráveis e outra com instâncias não vulneráveis.

Testes estatísticos são usados para verificar a validade de aferições feita sobre uma determinada população. O **p-value** (MOORE; NOTZ; FLIGNER, 2015) é usado para verificar a significância estatística dos resultados de um teste estatístico, é representado por um número entre 0 e 1 e interpretado da seguinte maneira:

- Menor *p-value* (normalmente ≤ 0.05) indica evidências fortes contra a hipótese nula, então você rejeita a hipótese nula;
- Maior *p-value* (> 0.05) indica evidência fraca contra a hipótese nula, então ela não é rejeitada

O **Wilcoxon Rank Sum Test** é um teste não paramétrico que compara dois grupos não pareados. É executado da seguinte forma (MOORE; NOTZ; FLIGNER, 2015):

- Ranqueia-se todos os valores, de baixo a cima, sem considerar o grupo a que cada valor pertence;
- Numera-se do menor para o maior número, de 1 a n, onde n é o número total de valores nos dois grupos;
- As médias de cada grupo são calculadas e se as médias do ranqueamento nos dois grupos forem estatisticamente diferentes, o *p-value* será menor que 0.05.

Trabalhar com estratégias de ranqueamento permite dispensar condições específicas sobre a forma da distribuição dos dados, como por exemplo, a distribuição normal (MOORE; NOTZ; FLIGNER, 2015). Então, as médias de cada grupo são ranqueadas em cada grupo e relatadas as duas médias. Se as médias do ranqueamento nos dois grupos forem estatisticamente diferentes, o *P-value* será menor que 0,05.

O que norteou as análises estatísticas acerca do histórico de mudanças das funções vulneráveis, foi a seguinte questão:

As métricas ODC são capazes de distinguir funções vulneráveis de funções não vulneráveis?

5.1.2 Dados Coletados

Para esta análise foram selecionados dois projetos, Mozilla (FOUNDATION, 2014) e Linux Kernel (CORPORATION, 2016), que se enquadram nos requisitos descritos na seção 4. Este processo resultou em uma base de dados a ser utilizada em nossas análises com informações sobre o histórico de mudanças em *funções*.

Ao analisar a disponibilidade das informações sobre as correções dos bugs de segurança, optou-se pela estratégia de coleta na *web*, descrita no Capítulo 4. Os repositórios da Mozilla Foundation Segurança Ad-(MFSA), e CVE Details (cvedetails.com) foram os *web reports* usados nesta coleta. Assim, foi possível determinar quais funções ou métodos foram reportados com bugs de segurança, e posteriormente corrigidos.

Para extrair o histórico de alterações, obtivemos os repositórios git de ambos os projetos (kernel, Mozilla) e desenvolveu-se um algoritmo automatizado seguindo os processos descritos na seção 4.2.

A base de dados provenientes da coleta do histórico de mudanças, pode-se encontrar no endereço (PIANCÓ; ANTUNES; FONSECA, 2016).

A partir dessas informações, classificou-se as mudanças de acordo com as métricas ODC, o resultado é apresentado na Tabela 4.

Tabela 4 – Número total de mudanças por métrica ODC

Metric	**Mozilla**	**Kernel**
NEC	506973	102259
NMC	557103	120685
NEFC	500304	107898
NMFC	545614	128965
NEM	79884	41596
NMM	67069	49215
NEV	359516	125534
NMV	355793	90057
# total	2972256	766209

5.1.3 Análise

No processo de análise, utilizou-se testes estatísticos para verificar se existiam diferenças significativas entre os valores das métricas ODC de *funções* vulneráveis e não vulneráveis. Como anteriormente, o teste estatístico não paramétrico *Wilcoxon Rank Sum Test*, foi utilizado, bem como, *boxplots*, onde avaliou-se a discrepância entre as instâncias vulneráveis e não vulneráveis, gerando também os *intervalos de confiança*.

5.1.4 Resultados

Aplicou-se então o teste de *Wilcoxon Rank-Sum* considerando duas hipóteses diferentes:

1. A diferença entre as médias dos dois grupos analisados é maior que 0 ;
2. A diferença entre as médias dos dois grupos analisados é inferior a 0;

A tabela 5 apresenta os valores de *p-value* resultantes do teste Wilcoxon para os dois projetos analisados. Os símbolos "+ "e '- ' representam os casos em que a hipótese 1 e 2 foram confirmadas, respectivamente.

Tabela 5 – *p-values* para o *Wilcoxon Rank Sum Test* usando o parâmetro "*greater*"

# p-value		
# metrics	Mozilla	Kernel
NEC	1.28E-08 (+)	1.04E-06 (+)
NMC	1.88E-06 (+)	2.17E-18 (+)
NEFC	1.18E-04 (+)	6.52E-04 (+)
NMFC	2.54E-09 (+)	3.86E-01 (+)
NEM	5.64E-53 (+)	4.13E-09 (-)
NMM	2.28E-65 (+)	7.49E-20 (-)
NEV	2.93E-01 (+)	1.66E-16 (+)
NMV	3.30E-04 (+)	3.62E-06 (+)

Como observa-se na tabela *Wilcoxon-Test*, a maioria das métricas escolhidas para representar as mudanças, se mostraram capazes de diferenciar o histórico de estruturas com e sem *report* para *bugs* de segurança. Apenas duas métricas para o projeto Kernel, NEM e NMM, se apresentaram contrárias às tendências de distinção para o histórico de mudanças.

Para uma análise mais detalhada, *boxplots* e *intervalos de confiança* (CIs) foram examinados. A Figura 10 apresenta os *boxplots* relacionados com os valores das métricas NEC e NMC para *funções* vulneráveis e não vulneráveis. Onde se observa que as medianas para *funções* com a vulnerabilidade é maior quando comparada às *funções* sem vulnerabilidades.

O mesmo pode ser observado na Figura 11, que representa os intervalos de confiança, para a média dos valores das mesmas métricas apresentados no gráfico dos *boxplots*, onde percebe-se intervalos diferentes para funções vulneráveis e não vulneráveis.

Os resultados, tanto para o teste de *Wilcoxon-Test*, quanto para as avaliações feitas nos *boxplots* e intervalos de confiança demonstram que o conjunto de métricas escolhidos para representar o histórico de mudanças permite distinguir as *funções* vulneráveis das não vulneráveis.

Mais detalhes acerca dos resultados e das bases de dados utilizadas nos processos de análises podem ser encontrados no endereço (PIANCÓ; ANTUNES; FONSECA, 2016).

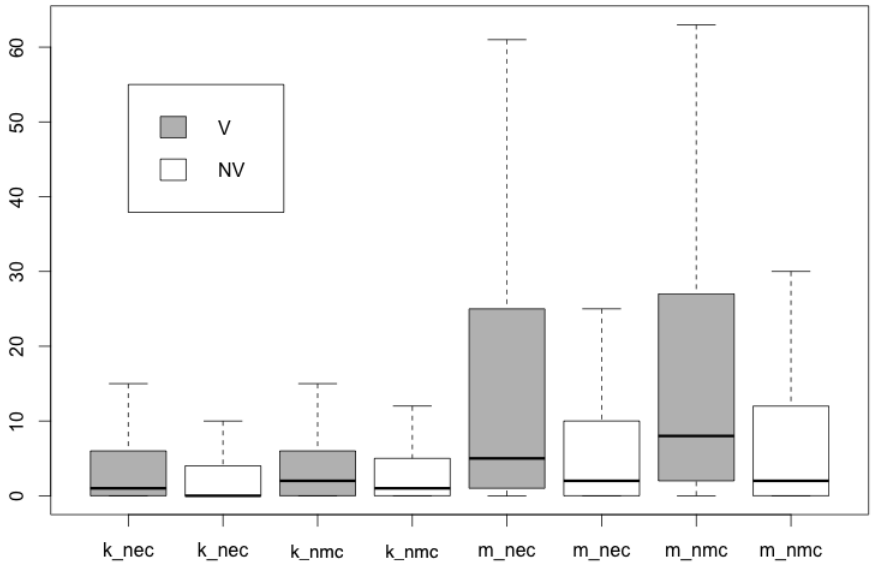

Figura 10 – *boxplots* para as métricas NEC e NMC em ambos os projetos, *Kernel* e *Mozilla* ("V-vulnerável, "NV-não vulnerável).

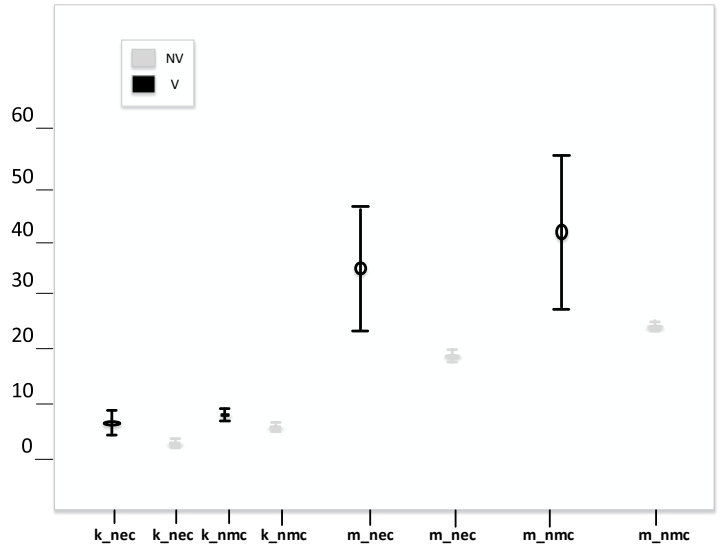

Figura 11 – Intervalos de Confiança por projeto

5.1.5 Síntese da Análise

Nesta análise foi constatado que as métricas de mudança representadas pela *ODC* quando analisadas numa perspectiva histórica, podem ser utilizadas para

discriminar um código com bugs de segurança, de um sem *bug* de segurança. Os resultados também mostram que algumas métricas não são eficazes para os tipos de relações que se quer encontrar (NEM, NMM).

5.2 ANALISANDO AS RELAÇÕES ENTRE O HISTÓRICO MUDANÇAS E *BUGS* NO SOFTWARE SOB UMA PERSPECTIVA DE TEMPORALIDADE

Análises sobre dados temporais são bastante úteis nas tarefas de criação de modelos de predição, classificação, e outros. Ao capturar dados ou eventos ao longo do tempo, são obtidos grandes quantidades de dados que ajudam a diminuir a taxa de erro, e podem melhorar os modelos de detecção ou predição (SILVA, 2016). Séries temporais podem ser descritas como coleções de informações de um fenômeno observado ao longo de um determinado período, organizadas em ordem cronológica (KIRCHGÄSSNER; WOLTERS; HASSLER, 2012).

Para avaliar, de maneira detalhada, a relação entre os tipos de mudanças feitas ao longo do tempo e bugs no software foi realizada uma análise com o propósito de encontrar quais tipos de mudanças podem melhor representar a similaridade entre históricos de mudanças em arquivos com bugs e sem bugs. Para tanto, foi usado uma técnica de classificação para séries temporais denominada de DTW (Dynamic Time Warping), que é uma das medidas mais adequadas para comparação entre séries temporais, possibilitando a identificação de padrões e a recuperação por conteúdo (RATANAMAHATANA; KEOGH, 2004)(BERNDT; CLIFFORD, 1994).

5.2.1 Tipos de Mudanças

Diferente da análise anterior, esta análise investigou mudanças feitas em arquivos *JAVA*, considerando uma gama maior dos tipos de mudanças investigados. Cada tipo de mudança representa um "nó"da AST em arquivos *JAVA*, de maneira que, cada tipo é determinado pelo sua especificação estabelecida na *API JDT/JAVA*. O uso desta *API* pode ser visto nos trabalhos de (KIM et al., 2006)(SOTO et al., 2016), onde auxilia no processo de identificação de tipos de códigos fonte. A relação de todos os

tipos de códigos que representam as mudanças feitas para esta análise pode ser vista no apêndice A.

Os tipos de mudanças nesta análise está diretamente relacionada aos tipos de códigos fonte suportados na linguagem *JAVA*. Com o auxílio da *API JDT* foi possível capturar quaisquer tipos de códigos fonte mapeados em uma *AST*. Dentre os tipos de códigos fonte capturados, existem alguns que são demasiadamente abstratos e poderiam confundir as análises, como por exemplo, o tipo *block*, que consiste em uma estrutura de código *JAVA*, que representa um sub-conjunto de outros tipos de códigos. Apenas os tipos de códigos fonte mais explícitos foram considerados, tais como: *method_invocation*, *if_statement* e outros. Dessa maneira, é possível analisar as mudanças, a partir de tipos de códigos fonte bem específicos, e que já mostraram ser bons atributos no processo de criação de modelos, como descritos em alguns estudos no Capítulo 3.

As mudanças estabelecidos nesta análise também podem ser classificados usando as mesmas diretrizes da adaptação *ODC* definidas na análise anterior. Optou-se pela não classificação das mudanças durante esta análise usando *ODC*, para que fosse feito uma análise quantitativa acerca da representatividade dos tipos de códigos fonte que melhor diferenciem mudanças ao longo do tempo entre históricos de mudanças em arquivos com e sem *report* de *bugs*.

5.2.2 *DTW - Dynamic Time-Warping*

Inicialmente desenvolvida para reconhecimento de palavras (BERNDT; CLIFFORD, 1994). *DTW* consiste no alinhamento ideal entre duas sequências (que dependem do tempo), e tem como principal objetivo encontrar séries temporais que apresentem comportamentos similares. De modo que, a dissimilaridade pode ser descrita como o não alinhamento entre séries temporais.

Existem evidências que comprovam a eficácia da técnica *DTW* no processo de dissimilaridade em séries temporais, gerando excelentes resultados em muitas aplicações. Porém, quando sua utilização envolve grandes quantidades de dados a tarefa de classificação para dissimilaridade, ou similaridade entre séries temporais pode se tornar algo dispendioso (SILVA, 2014).

5.2.3 Análise da Similaridade entre Mudanças

Para esta análise, adotou-se uma abordagem diferente da usada na seção anterior. Esta, consiste na investigação acerca da temporalidade das mudanças, entre arquivos com *bugs* e sem *bugs*. É analisado se a partir de um sub-conjunto de tipos de mudanças, é possível distinguir as séries temporais, produzidas a partir dos diferentes históricos de mudanças. O projeto escolhido para realizar esta análise foi o *tomcat*, pois, além da relevância que o projeto tem na comunidade de desenvolvimento, este dispõe de informações de fácil acesso para que experimentos ou estudos possam ser realizados.

Ao averiguar os dados produzidos durante a coleta do histórico de mudanças no projeto *tomcat*, percebeu-se que os dados não apresentavam uma distribuição normal (MOORE; NOTZ; FLIGNER, 2015), e que os históricos de mudanças tinham tamanhos diferentes para arquivos com e sem *bugs*. Considerando estas duas premissas, investigou-se qual seria a melhor técnica para comparar séries temporais, e se era possível estabelecer diferenças significativas entre históricos de mudanças diferentes, optando-se pela técnica DTW.

No processo de análise sobre a similaridade entre as séries temporais, assumiu-se que, quanto maior a taxa de similaridade entre as séries, menor a taxa de diferenciação entre elas. Essa diferenciação é assumida como sendo a taxa de dissimilaridade. Nesta segunda análise, investiga-se quais tipos de códigos fonte (*JAVA*), associados às mudanças ao longo do tempo, são capazes de melhor diferenciar séries temporais entre arquivos com e sem report de bugs.

O processo de coleta do histórico de mudanças no projeto *tomcat* resultou em um *data set* contendo 222.719 alterações em arquivos. Destas, 189.639 representam mudanças feitas em arquivos com *reports* de *bugs* e 33.080 em arquivos sem estes *reports*. As mudanças foram representadas pelos tipos de códigos fonte definidos pela *API JDT*, como descrito na seção 5.2.1. A grande quantidade de tipos de códigos, exatamente 92 tipos, representam um desafio na tarefa de comparação entre séries temporais. Para tanto, foram definidos seleções específicas para diminuir a dimensionalidade do *data set* para tornar viável uma análise sobre o histórico de mudanças. Os 92 tipos de códigos fonte definidos pela *API JDT* para representar os códigos fonte

JAVA são mostrados no Apêndice A.

A primeira seleção foi definida de maneira que, apenas os tipos que tinham uma média de modificações ao longo do tempo maior que 0 iriam para a próxima fase de seleção. Deste processo restaram apenas 69 tipos de códigos fonte. Para a segunda seleção, escolheu-se executar uma estratégia de seleção de atributos mais sofisticada, denominada de *Information Gain* ou entropia, que consiste na caracterização das impureza nos dados. É uma medida da falta de homogeneidade dos dados de entrada em relação a sua classificação (GAMA, 2002).

Desta segunda seleção restaram apenas 37 tipos de códigos fonte que representam os melhores atributos no *data set* para o processo de classificação das séries temporais a serem usados na *DTW*. Entretanto, em um *data set* desta dimensão ainda é inviável a execução de classificação de séries temporais usando o *DTW*. Para tanto, analisou-se outra técnica de seleção de atributos, que baseia-se na escolha das formas mais representativas de dados, a partir de combinações lineares dos atributos originais. Esta técnica é denominada de PCA (*Principal Component Analysis*), na qual a representação do *data set* é feita através de *auto vetores*, que são significativos no reconhecimento dos principais atributos a serem usados.

De modo a viabilizar a análise proposta nesta seção, foram escolhidos os atributos (tipos de códigos fonte) que representassem as mudanças no *data set*, de modo que, foi considerada a interceção entre os melhores atributos advindos dos dois componentes principais na seleção de atributos *PCA*, e os atributos advindos da seleção através do algoritmo de ganho de informação (*Information Gain*), restando apenas 10 tipos de códigos fonte para a execução da classificação sobre séries temporais usando o *DTW*. Os tipos de códigos fonte associados às mudanças que restaram após os filtros de seleção de atributos podem ser vistos na primeira coluna da Tabela 6.

A pergunta que conduziu as análises feitas sobre as mudanças ao longo do tempo, considerando a similaridade entre séries temporais foi:

É possível identificar dissimilaridades entre as séries temporais que representam o histórico de mudanças em arquivos com bug, dos históricos de arquivos sem bugs, a partir de um subconjunto específico de tipos de mudanças?

5.2.4 Resultados

Das análises feitas na seção anterior, obteve-se as distâncias que determinaram as similaridades entre as séries temporais, para cada tipo de código, como visto na Tabela 6. Foram consideradas as medianas das distâncias totais de cada combinação entre as séries temporais que representaram os históricos de mudanças em arquivos com e sem *report* de *bugs*. A mediana foi escolhida para representar os valores das distâncias totais, pois ela não se influencia por valores erráticos e não significativos.

Tabela 6 – Resultados da Classificação por Similaridade usando *DTW*

Tipo de Código Fonte	Similaridade DTW
VARIABLE_DECLARATION_FRAGMENT	**0,75**
VARIABLE_DECLARATION_STATEMENT	**0,6326**
IF_STATEMENT	**1,01**
INFIX_EXPRESSION	**1,1974**
BLOCK	1,1516
EXPRESSION_STATEMENT	1,2830
SWITCH_CASE	N/A
FOR_STATEMENT	N/A
METHOD_INVOCATION	**1,8**
SIMPLE_NAME	5,07

Mesmo com o uso dos filtros para reduzir a dimensão de representação dos dados, observa-se que na Tabela 6, os tipos de códigos fonte "SWITCH_CASE"e "FOR_-STATEMENT"não foram capazes de gerar valores que representassem a similaridade entre as séries temporais. Isso se deu pois se fez necessário estabelecer um limite mínimo de quantidades de alterações em arquivos, para a execução do algoritmo de similaridade *DTW*. Limite este, que foi estabelecido considerando a média de todas as alterações feitas nos arquivos do *data set*, que resultou em um valor de 16 alterações em média por arquivo. Com isso, ao investigar as instâncias que continham os históricos de mudanças sem os reports de *bugs*, constatou-se que, para os dois tipos de códigos fonte supracitados, não haviam mais que 13 modificações feitas nestes históricos de mudanças para estes tipos.

Como mencionado anteriormente, só seriam considerados como resultados que pudessem ser relevantes, os tipos de códigos fonte que tem o menor nível de

granularidade, excluindo assim, os tipos mais genéricos. Assim, para calcular a média de similaridade entre as séries temporais que representam os históricos de mudanças, excluímos os seguintes tipos: "BLOCK"e "EXPRESSION_STATEMENT", que são tipos que podem conter blocos inteiros de código fonte, que por sua vez, pode conter uma série de outros tipos; "SIMPLE_NAME", que representa um identificador, que pode ser de uma variável, de um objeto, ou assumir valores booleanos e até mesmo *null literal*, assim, esse tipo não detêm de um nível aceitável de granularidade que esta análise pede.

Considerando os tipos de códigos que restaram após todas as eliminações, a distância que determina a similaridade entre as séries temporais foi de 1.076, para os históricos de mudanças em arquivos com e sem *reports* de *bugs*. Lembrando que, segundo a descrição de similaridade, quanto mais próximo do valor "0", maior será a similaridade entre as séries temporais. De modo que, os valores quanto mais distantes de "0"representam a dissimilaridade entre as séries temporais. É possível observar na Figura 12, as representações das séries temporais em arquivos com *report* de *bugs* (gráficos superiores) e arquivos sem *(*report) de *bugs* (gráficos inferiores), para três dos tipos de códigos fonte que foram discriminantes na comparação entre séries temporais.

Assim, podemos supor que o conjunto de tipos de códigos restantes podem ser úteis na diferenciação entre históricos de mudanças em arquivos com *bugs*, dos históricos de mudanças em arquivos sem *bugs*. Mais estudos precisam ser feitos para que se tenha uma maior certeza da afirmação anterior. Visto que, estas análises foram aplicadas a um único domínio (*tomcat*).

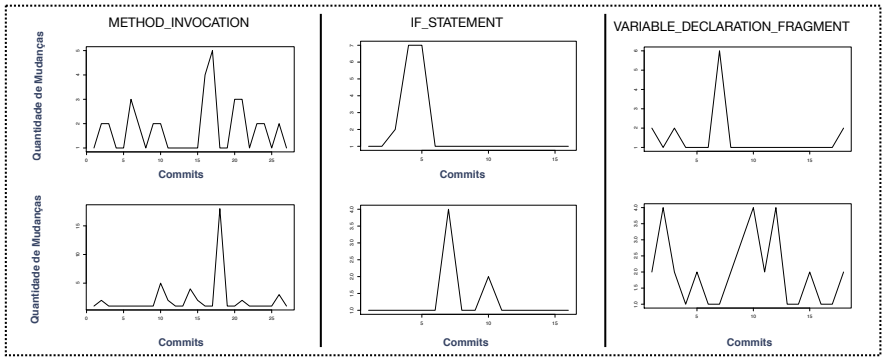

Figura 12 – Representação das Diferentes Séries Temporais entre arquivos com *report* de *bugs*, e arquivos sem *report*.

6 CONCLUSÃO

Neste trabalho foram feitas diversas análises a partir das mudanças em códigos fonte. Mais precisamente, estudou-se a relação entre mudanças e a ocorrência de bugs no software.

Para entender melhor as mudanças em códigos fonte, foram também estudadas algumas taxonomias de classificação de mudanças, de modo a introduzir uma semântica associada a cada mudança. A taxonomia que melhor representou as mudanças extraídas foi a *Orthogonal Defect Classification* (*ODC*), que já é muito difundida e possui um vasto suporte teórico. Na análise feita no Capítulo 5, a classificação das mudanças pela *ODC* foi bem-sucedida. Pois, foi possível estabelecer, através de testes estatísticos, diferenças entre históricos de mudanças em funções e métodos reportados como vúlneraveis, dos que não foram reportados como vulneraveis.

Já na segunda análise descrita no Capítulo 5 foram especificados os tipos de códigos fonte (*JAVA*) para representar as mudanças em arquivos. Nesta análise o auxílio de técnicas para seleção de atributos (*PCA*, *Information Gain*) foi crucial, proporcionando uma diminuição na dimensão dos atributos do *data set*, resultando em um menor conjunto de tipos de códigos fonte a serem discriminados na tarefa de comparação entre séries temporais sobre os históricos de mudanças. Nesta análise foi possível estabelecer que é possível diferenciar uma série temporal com *report* de *bug* de uma que não teve *report* de *bug*. Isso pode ajudar na tarefa de sugestão de alarmes, para prováveis *bugs* durante a fase de desenvolvimento do software.

Também foi produzido um mecanismo para coletar as mudanças em que arquivos que são movidos para outras partes do projeto têm seu histórico de mudanças preservado, diferenciando de alguns trabalhos mencionados no Capítulo 3. Pois, nos casos em que arquivos que são movidos para outras partes do projeto, seu histórico de mudanças será preservado. Esse mecanismo pode diminuir a taxa de erros durante a geração de modelos de predição e deteção de *bugs*. Também deve-se levar em conta que neste mecanismo os *bugs* são diferenciados das *issues*, que em muitos estudos

são coletados como sendo elementos iguais.

Outra vantagem no desenvolvimento das ferramentas de coleta de mudanças sobre *bugs* no software, foi a elaboração de uma base de dados contendo uma grande quantidade de dados que podem ser usados em diversas abordagens já existentes. Isso é possível porque as informações contidas nesta base de dados tem mais riqueza de detalhes do que nas bases de dados disponibilizadas pelas abordagens descritas no Capítulo 3. Alem disso, com as ferramentas de coleta descritas no Capítulo 4 é possível realizar análises com um menor grau de granularidade, pois a partir dos dados dispostos no *data set* produzido, é possível se chegar a estruturas atômicas em arquivos de códigos fonte, tais como: classes, métodos e funções.

As análises acerca das relações entre mudanças e *bugs* no software ainda precisam ser estudadas mais detalhadamente, uma vez que são as mudanças em códigos fonte que permitem que sistemas computacionais se mantenham funcionais. Evitar bugs *bugs* durante estas mudanças pode trazer impactos positivos às empresas e organizações.

6.1 TRABALHOS FUTUROS

Tendo em vista que as análises foram feitas em contextos bem específicos, é preciso investigar novos projetos e tentar validar os resultados já obtidos. Pode-se também explorar outras caracterizações de mudanças, além da *ODC*, para avaliar se a representatividade da *ODC* ainda permaneceria.

Uma análise que pode ser feita sobre os dados já coletados, é investigação acerca de prováveis padrões de codificação, que possam estar associados à ocorrência de *bugs* no software. Ainda que alguns estudo já tenham sido feitos acerca destes padrões, na maioria dos casos, os experimentos são feitos em pequenos conjuntos de dados, ou em domínios bem específicos. Os dados gerados no Capítulo 4, poderiam aumentar as probabilidade de encontrar padrões que representassem diferentes contextos, para mais de um domínio de análise.

Existe também a possibilidade de aprimorar a ferramenta de coleta de mudanças proposto neste trabalho, adaptando o mecanismo de coleta a outros repositórios de códigos fonte, tais como: *mercurial*, *svn*. Dessa maneira, seria possível extrair

grandes quantidades de dados sobre as mudanças em códigos fonte.

Pode-se também tentar aplicar abordagens de outros trabalhos sobre os dados aqui extraídos, seja para predição ou detecção de ugs, uma vez que já existem trabalhos que analisaram alguns dos projetos que foram usados para a extração de mudanças neste trabalho. Com isso, seria possível comparar alguns resultados obtidos por outros estudos com os já obtidos aqui.

6.2 AMEAÇAS A VALIDADE DO TRABALHO

Um dos principais desafios em pesquisas ou experimentos que utilizam dados sobre mudanças em códigos fonte é realizar as extrações dessas mudanças, pois o tempo estimado para extrai-las torna inviável, em muitos casos, o uso de algumas abordagens que tentam produzir modelos de predição e detecção de *bugs* no software, a depender do tamanho do sistema.

Este trabalho se limitou na extração dos dados sobre mudanças em apenas quatro projetos para realizar as análises, pois o tempo para realizar tais análises, o que pode ter influenciado negativamente alguns resultados gerando possíveis acidentes estatísticos. Outro fator relevante que pode ter influenciado as análises feitas foi a restrição nas linguagens de programação em que os projetos eram escritos, *Java C,C++*, o que pode ter limitado o escopo de análise sobre as relações entre bugs e mudanças.

REFERÊNCIAS

BERNDT, D. J.; CLIFFORD, J. Using dynamic time warping to find patterns in time series. In: SEATTLE, WA. **KDD workshop**. [S.l.], 1994. v. 10, n. 16, p. 359–370.

BINH VIET HOANG ANH, T. P. X. A. T. L. T. C/c++ source code analyzing with cdt. **Proceedings of the Second Asia Pacific International Conference on Information Science and Technology**, p. 6, 2009.

BOSU, A. et al. Identifying the characteristics of vulnerable code changes: An empirical study. In: **22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2014.

CASEY, V.; RICHARDSON, I. Implementation of global software development: a structured approach. **Software Process: Improvement and Practice**, Wiley Online Library, v. 14, n. 5, p. 247–262, 2009.

CHILLAREGE, R. et al. Orthogonal defect classification-a concept for in-process measurements. **Software Engineering, IEEE Transactions on**, IEEE, v. 18, n. 11, p. 943–956, 1992.

CHRISTEY, S. Unforgivable vulnerabilities. **Black Hat Briefings**, 2007.

CLARKE, P.; O'CONNOR, R. V. The influence of spi on business success in software smes: An empirical study. **Journal of Systems and Software**, v. 85, n. 10, p. 2356 – 2367, 2012. ISSN 0164-1212. Automated Software Evolution. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121212001355>.

CLARKE, P.; O'CONNOR, R. V.; LEAVY, B. A complexity theory viewpoint on the software development process and situational context. In: **2016 IEEE/ACM International Conference on Software and System Processes (ICSSP)**. [S.l.: s.n.], 2016. p. 86–90.

CORPORATION, M. **Download Linux Kernel - Vulnerabilities Information**. 2016. Available in http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html. Last access in March 2016.

COUTO, C. et al. Predicting software defects with causality tests. **Journal of Systems and Software**, Elsevier, v. 93, p. 24–41, 2014.

DURAES, J.; MADEIRA, H. Definition of software fault emulation operators: A field data study. In: IEEE. **Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on**. [S.l.], 2003. p. 105–114.

DURAES, J. A.; MADEIRA, H. S. Emulation of software faults: A field data study and a practical approach. **IEEE Transactions on Software Engineering**, IEEE, v. 32, n. 11, p. 849–867, 2006.

ELECTRICAL, I. of; ENGINEERS, E. **IEEE Standard Dictionary of Measures to Produce Reliable Software**. 1989.

FLURI, B. et al. Change distilling: Tree differencing for fine-grained source code change extraction. **IEEE Transactions on Software Engineering**, IEEE, v. 33, n. 11, 2007.

FOUNDATION, M. **Mozilla**. 2014. Http://www.mozilla.org/. Last access in March 2014.

GAMA, J. Árvores de decisão. **Palestra ministrada no Núcleo da Ciência de Computação da Universidade do Porto, Porto**, 2002.

GEGICK, M.; ROTELLA, P.; XIE, T. Identifying security bug reports via text mining: An industrial case study. In: IEEE. **Mining software repositories (MSR), 2010 7th IEEE working conference on**. [S.l.], 2010. p. 11–20.

HAMANO, J.; TORVALDS, L. **Git-fast version control system**. 2005.

HERZIG, K.; ZELLER, A. The impact of tangled code changes. In: IEEE. **Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on**. [S.l.], 2013. p. 121–130.

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.

JIMENEZ, W.; MAMMAR, A.; CAVALLI, A. Software vulnerabilities, prevention and detection methods: A review1. **Security in Model-Driven Architecture**, p. 6, 2009.

KIM, S.; JR, E. J. W.; ZHANG, Y. Classifying software changes: Clean or buggy? **IEEE Transactions on Software Engineering**, IEEE, v. 34, n. 2, p. 181–196, 2008.

KIM, S. et al. Automatic identification of bug-introducing changes. In: IEEE. **Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on**. [S.l.], 2006. p. 81–90.

KIRCHGÄSSNER, G.; WOLTERS, J.; HASSLER, U. **Introduction to modern time series analysis**. [S.l.]: Springer Science & Business Media, 2012.

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

MCGRAW, G. Software security. **IEEE Security & Privacy**, IEEE, v. 2, n. 2, p. 80–83, 2004.

MOLYNEAUX, I. **The art of application performance testing: Help for programmers and quality assurance**. [S.l.]: "O'Reilly Media, Inc.", 2009.

MOORE, D. S.; NOTZ, W. I.; FLIGNER, M. A. **The Basic Practice of Statistics**. 7nd. ed. New York, NY, USA: W. H. Freeman & Co., 2015. ISBN 146414253.

NEUHAUS, S. et al. Predicting vulnerable software components. In: **Proceedings of the 14th ACM conference on Computer and Communications Security (CCS'07)**. New York, New York, USA: ACM Press, 2007. p. 529–540. ISBN 9781595937032.

NISTOR, A.; JIANG, T.; TAN, L. Discovering, reporting, and fixing performance bugs. In: IEEE. **Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on**. [S.l.], 2013. p. 237–246.

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

PALOMBA, F. et al. Detecting bad smells in source code using change history information. In: IEEE. **Automated software engineering (ASE), 2013 IEEE/ACM 28th international conference on**. [S.l.], 2013. p. 268–278.

PAN, K.; KIM, S.; WHITEHEAD, E. J. Toward an understanding of bug fix patterns. **Empirical Software Engineering**, Springer, v. 14, n. 3, p. 286–315, 2009.

PIANCÓ, M.; ANTUNES, N.; FONSECA, B. **A Dataset of Code Change History and Software Vulnerabilities**. 2016. Disponível em: <https://eden.dei.uc.pt/~nmsa/changes-dataset>.

RATANAMAHATANA, C. A.; KEOGH, E. Making time-series classification more accurate using learned constraints. In: SIAM. **Proceedings of the 2004 SIAM International Conference on Data Mining**. [S.l.], 2004. p. 11–22.

SHAR, L. K.; TAN, H. B. K. Predicting common web application vulnerabilities from input validation and sanitization code patterns. In: IEEE. **Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on**. [S.l.], 2012. p. 310–313.

SHIHAB, E. et al. High-impact defects: a study of breakage and surprise defects. In: ACM. **Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering**. [S.l.], 2011. p. 300–310.

SHIN, Y. et al. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. **Software Engineering, IEEE Transactions on**, IEEE, v. 37, n. 6, p. 772–787, 2011.

SILVA, A. Maggioni e. **Classificação de séries temporais baseada em análise de recorrência e extração de características**. Dissertação (Mestrado), 2016.

SILVA, D. F. **Classificação de séries temporais por similaridade e extração de atributos com aplicação na identificação automática de insetos**. Tese (Doutorado) — Universidade de São Paulo, 2014.

SOTO, M. et al. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In: ACM. **Proceedings of the 13th International Workshop on Mining Software Repositories**. [S.l.], 2016. p. 512–515.

TICHY, W. F. Rcs—a system for version control. **Software: Practice and Experience**, Wiley Online Library, v. 15, n. 7, p. 637–654, 1985.

WOHLIN, C. et al. **Experimentation in software engineering**. [S.l.]: Springer, 2012. ISBN 978-3-642-29043-5.

YOON, Y.; MYERS, B. A.; KOO, S. Visualization of fine-grained code change history. In: IEEE. **Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on**. [S.l.], 2013. p. 119–126.

ZEGEYE, L.; SAILIO, M. Vulnerability database analysis for 10 years for ensuring security of cyber critical green infrastructures. In: IEEE. **AFRICON, 2015**. [S.l.], 2015. p. 1–5.

Apêndices

APÊNDICE A – MÉTRICAS DE MUDANÇA

Estruturas de códigos fonte *JAVA*, que foram os tipos de mudanças (Métricas) especificados pela *API JDT*.

Estruturas de Códigos Fonte *Java*		
ARRAY_ACCESS	METHOD_DECLARATION	WHILE_STATEMENT
ARRAY_CREATION	METHOD_INVOCATION	INSTANCEOF_EXPRESSION
ARRAY_INITIALIZER	NULL_LITERAL	LINE_COMMENT
ARRAY_TYPE	NUMBER_LITERAL	BLOCK_COMMENT
ASSERT_STATEMENT	PACKAGE_DECLARATION	TAG_ELEMENT
ASSIGNMENT	PARENTHESIZED_EXPRESSION	TEXT_ELEMENT
BLOCK	POSTFIX_EXPRESSION	MEMBER_REF
BOOLEAN_LITERAL	PREFIX_EXPRESSION	METHOD_REF
BREAK_STATEMENT	PRIMITIVE_TYPE	METHOD_REF_PARAMETER
CAST_EXPRESSION	QUALIFIED_NAME	ENHANCED_FOR_STATEMENT
CATCH_CLAUSE	RETURN_STATEMENT	ENUM_DECLARATION
CHARACTER_LITERAL	SIMPLE_NAME	ENUM_CONSTANT_DECLARATION
CLASS_INSTANCE_CREATION	SIMPLE_TYPE	TYPE_PARAMETER
COMPILATION_UNIT	SINGLE_VARIABLE_DECLARATION	PARAMETERIZED_TYPE
CONDITIONAL_EXPRESSION	STRING_LITERAL	QUALIFIED_TYPE
CONSTRUCTOR_INVOCATION	SUPER_CONSTRUCTOR_INVOCATION	WILDCARD_TYPE
CONTINUE_STATEMENT	SUPER_FIELD_ACCESS	NORMAL_ANNOTATION
DO_STATEMENT	SUPER_METHOD_INVOCATION	MARKER_ANNOTATION
EMPTY_STATEMENT	SWITCH_CASE	SINGLE_MEMBER_ANNOTATION
EXPRESSION_STATEMENT	SWITCH_STATEMENT	MEMBER_VALUE_PAIR
FIELD_ACCESS	SYNCHRONIZED_STATEMENT	ANNOTATION_TYPE_DECLARATION
FIELD_DECLARATION	THIS_EXPRESSION	ANNOTATION_TYPE_MEMBER_DECLARATION
FOR_STATEMENT	THROW_STATEMENT	MODIFIER
IF_STATEMENT	TRY_STATEMENT	UNION_TYPE
IMPORT_DECLARATION	TYPE_DECLARATION	DIMENSION
INFIX_EXPRESSION	TYPE_DECLARATION_STATEMENT	LAMBDA_EXPRESSION
INITIALIZER	TYPE_LITERAL	INTERSECTION_TYPE
JAVADOC	VARIABLE_DECLARATION_EXPRESSION	NAME_QUALIFIED_TYPE
LABELED_STATEMENT	VARIABLE_DECLARATION_FRAGMENT	CREATION_REFERENCE
SUPER_METHOD_REFERENCE	VARIABLE_DECLARATION_STATEMENT	EXPRESSION_METHOD_REFERENCE