Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Bacharelado em Engenharia de Software

João Eduardo Ribeiro de Medeiros

Estudo Comparativo de Ferramentas de Análise Estática de Código

Natal-RN

2017

João Eduardo Ribeiro de Medeiros

Estudo Comparativo de Ferramentas de Análise Estática de Código

Trabalho de Conclusão de Curso apresentado ao Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do grau de bacharel em Engenharia de Software.

Orientador: Dr. Umberto Souza da Costa

Natal-RN

2017

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Setorial Prof. Ronaldo Xavier de Arruda - CCET

Medeiros, João Eduardo Ribeiro de.
Estudo comparativo de ferramentas de análise estática de código / João Eduardo Ribeiro de Medeiros. - Natal, 2017.
72f.: il.

Monografia (graduação) - Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Bacharelado em Engenharia de Software.
Orientador: Umberto Souza da Costa.

1. Engenharia de software. 2. Análise estática. 3. Qualidade de código. 4. Detecção de bugs de software. 5. Otimização de código. I. Costa, Umberto Souza da. II. Título.

RN/UF/CCET CDU 004.41

Monografia de Graduação sob o título "Estudo Comparativo de Ferramentas de Análise Estática de Código"apresentada por João Eduardo Ribeiro de Medeiros e aceita pelo Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte, sendo aprovada por todos os membros da banca examinadora abaixo especificada:

Dr. Umberto Souza da Costa
Orientador
Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte

Dr. Everton Ranielly de Sousa Cavalcante
Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte

Dra. Roberta de Souza Coelho
Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte

Natal-RN, 24 de novembro de 2017.

A minha família e amigos que sempre me apoiaram.

Agradecimentos

Agradeço primeiramente aos meus pais, por todo o apoio não só durante a graduação mas durante toda a vida. Sou grato também aos amigos que conheci durante a graduação por terem me ajudado a me tornar quem eu sou hoje.

Sou profundamente grato por todos aqueles que tive a oportunidade de trabalhar durante o curso, principalmente aos meus amigos Danilo, Alexis, Thaian, Ciro, Gustavo, Pedro e Stefano que sempre estiveram comigo nos momentos mais difíceis da graduação.

Agradeço também ao Gabriel Menegatti, dono da empresa *Simbiose Ventures*, onde trabalho atualmente, não só por ter cedido todas as informações necessárias para que o estudo pudesse ser realizado sobre o projeto *SlicingDice* mas também por ter sido completamente flexível todas as vezes que precisei faltar para me concentrar nos trabalhos da faculdade.

Por fim mas não menos importante, agradeço ao meu orientador Umberto Souza da Costa por toda a orientação durante o desenvolvimento dessa monografia e por todos os ensinamentos passados que, com certeza, foram/serão muito importantes para o meu desenvolvimento acadêmico e profissional.

Obrigado.

"Não sabendo que era impossível, foi lá e fez."

Jean Cocteau

Resumo

A análise estática de código está cada vez mais popular entre os desenvolvedores de sistemas devido aos diversos benefícios que ela traz à produção de software e à crescente necessidade de produção de software de qualidade. Neste cenário, diversas ferramentas de análise estática vêm surgindo e, com tantas opções, torna-se necessário avaliá-las e compará-las, a fim de entender melhor seus benefícios e auxiliar o desenvolvedor a escolher a ferramenta mais adequada a seu projeto. Neste Trabalho de Conclusão de Curso, discutimos a análise estática de código no contexto das linguagens de programação C, Java e Python. Primeiro, investigamos diversas ferramentas de análise estática e, então, comparamos sua eficiência aplicando-as a projetos de software reais. Desta forma, este trabalho pretende comparar ferramentas existentes no mercado, identificar as melhores opções disponíveis e indicar qual das linguagens abordadas é a provida de melhores recursos de análise estática.

Palavras-chave: análise estática, qualidade de código, detecção de bugs de software, otimização de código.

Abstract

Static code analysis is becoming more popular among system developers due to the many benefits it brings to the software development and to the growing need of developing better software. In this scenario, several static analysis tools have been emerging and, with so many options, it is necessary to evaluate and compare them, aiming at better understanding they benefits and to assist the developer in choosing the most appropriate tool to their projects. In this work, we discuss static code analysis in the context of C, Java and Python programming languages. First, we investigate several static analysis tools and then compare their efficiency by applying them to real software projects. In this way, this work intends to compare existing tools at the market, to identify the best available options and to indicate which of the covered languages is the one provided with the best static analysis features.

Keywords: static analysis, code quality, software bug detection, code optimization.

Lista de figuras

Figura 1 Quantidade de padrões de bugs detectados nas ferramentas C. . . 44

Figura 2 Interface gráfica da ferramenta *SpotBugs*. 51

Figura 3 Exemplo de resposta da ferramenta Bandit. 54

Figura 4 Quantidade de padrões de *bugs* para as ferramentas Java. 55

Figura 5 Quantidade de padrões de *bugs* para as ferramentas Python. . . 56

Figura 6 Quantidade de padrões de *bugs* para as ferramentas C. 57

Figura 7 Uso de memória nas ferramentas PMD, Checkstyle, FindBugs e SpotBugs. 59

Figura 8 Tempo de execução ferramentas PMD e CheckStyle (por projeto). 60

Figura 9 Tempo de execução ferramentas FindBugs e SpotBugs (por projeto). 60

Figura 10 Tempo de execução ferramentas Java (por quantidade de linhas). 61

Figura 11 Tempo de execução ferramentas Java (por quantidade de padrões de *bugs*). 61

Figura 12 Uso de memória nas ferramentas Pylint, Pyflakes, Bandit e pycodestyle. 63

Figura 13 Tempo de execução ferramentas Python (por projeto). 63

Figura 14 Tempo de execução ferramentas Python (por linhas de código). 64

Figura 15 Tempo de execução ferramentas Python (por quantidade de padrões de bugs). 64

Figura 16 Uso de Memória nas ferramentas CppCheck e FlawFinder. . . . 65

Figura 17 Tempo de execução ferramentas C (por projeto). 66

Figura 18 Tempo de execução ferramentas C (por linhas de código). 66

Figura 19 Tempo de execução ferramentas Python (por quantidade de padrões de bugs). 67

Lista de tabelas

Tabela 1 Vantagens e desvantagens da análise estática. 24

Tabela 2 Vantagens e desvantagens da análise dinâmica. 25

Tabela 3 Classificação de ferramentas Java. 26

Tabela 4 Classificação de ferramentas Python. 27

Tabela 5 Classificação de ferramentas C. 27

Tabela 6 Comparação entre os trabalhos relacionados. 38

Tabela 7 Conjunto de regras utilizado no estudo comparativo 1. 41

Tabela 8 Resultados do estudo 1 para as ferramentas C. 42

Tabela 9 Diferenças entre os resultados do artigo e os obtidos nessa monografia. 44

Tabela 10 Resultados do estudo comparativo 1 para as ferramentas Java. . 45

Tabela 11 Resultados do estudo comparativo 1 para as ferramentas Python. 46

Tabela 12 Ferramentas de análise estática utilizadas no estudo comparativo 2. 47

Tabela 13 Programas analisados para a linguagem Java. 49

Tabela 14 Programas analisados para a linguagem Python. 49

Tabela 15 Programas analisados para a linguagem C. 50

Tabela 16 Comparação da interface com o usuário das ferramentas Java. . 52

Tabela 17 Comparação da interface com o usuário das ferramentas Python. 52

Tabela 18 Comparação da interface com o usuário das ferramentas C. . . . 52

Tabela 19 Nível de detalhamento dos padrões de *bugs* apresentados pelas ferramentas Java. 53

Tabela 20 Nível de detalhamento dos padrões de *bugs* apresentados pelas ferramentas Python. 53

Tabela 21 Nível de detalhamento dos padrões de *bugs* apresentados pelas ferramentas C. 54

Tabela 22 Tempo de execução e uso de memória nas ferramentas PMD e CheckStyle. 58

Tabela 23 Tempo de execução e uso de memória ferramentas FindBugs e SpotBugs. 58

Tabela 24 Tempo de execução e uso de memória ferramentas Pylint e Pyflakes. 62

Tabela 25 Tempo de execução e uso de memória ferramentas Bandit e pycodestyle. 62

Tabela 26 Tempo de execução e uso de memória ferramentas CppCheck e FlawFinder. 65

Sumário

1 Introdução 15

1.1 Motivação . . . 16

1.2 Objetivos . . . 17

1.2.1 Gerais . . . 17

1.2.2 Específicos . . . 17

2 Embasamento Teórico 18

2.1 Processo de desenvolvimento de *software* . . . 18

2.2 Falhas comuns no *software* . . . 19

2.2.1 *Buffer Overflow* . . . 19

2.2.2 SQL *Injection* . . . 19

2.2.3 Outras Falhas . . . 20

2.3 Teste de *Software* . . . 21

2.4 Análise de programas . . . 21

2.5 Classificação de analisadores estáticos . . . 25

3 Trabalhos relacionados 29

3.1 An overview on the Static Code Analysis approach in Software Development . . . 29

3.2 A Comparative Study of Industrial Static Analysis Tools . . . 30

3.3 Test-driving static analysis tools in search of C code vulnerabilities . . . 32

3.4 Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools . . . 34

3.5 Um estudo sobre a correlação entre defeitos de campo e *warnings* reportados por uma ferramenta de análise estática 36

3.6 Considerações acerca da análise inicial 37

4 Estudo Comparativo 39

4.1 Objetivos 39

4.2 Ambiente do Estudo Comparativo 39

4.3 Estudo Comparativo 1 40

4.3.1 Resultados do Estudo Comparativo 42

4.3.2 Considerações 46

4.4 Estudo Comparativo 2 47

4.4.1 Programas analisados 48

4.4.2 Execução dos Estudos 50

4.4.3 Resultado do Estudo 51

4.4.3.1 Interface com o usuário 51

4.4.3.2 Nível de detalhamento dos padrões de *bugs* 53

4.4.3.3 Quantidade de padrões de *bugs* 54

4.4.3.4 Tempo de execução e consumo de memória 57

4.4.4 Considerações 67

4.5 Conclusão 68

5 Considerações finais 69

Referências 71

1 Introdução

A evolução da tecnologia, principalmente na área da Computação, possibilitou que mais e mais pessoas tivessem a oportunidade de aprender programação e a escrever seu próprio programa/algoritmo. Diante dessa evolução, podemos perceber um crescimento na quantidade de programadores inexperientes e na quantidade de programas cada vez mais complexos. Além disso erros sutis são cada vez mais comuns e na maioria das vezes passam despercebidos até pelos programadores mais experientes.

Nesse contexto, precisamos cada vez mais de ferramentas que apoiem o desenvolvimento de software, sendo a análise estática uma das técnicas que podem ajudar o programador. A análise estática é realizada sobre o código fonte ou sobre o código objeto e tem por objetivo identificar padrões no código que podem causar falhas, problemas de segurança, problemas de performance, etc., enquanto que a análise dinâmica é realizada sobre o código em execução e tem por objetivo identificar, principalmente, se o código está funcionando corretamente para um conjunto limitado de entradas.

A análise estática permite que diversos erros sejam encontrados antes mesmo que o programa tenha que ser compilado (somente válido para as ferramentas que analisam somente o código fonte). No entanto, é recomendado que ambas as análises sejam utilizadas em conjunto durante o desenvolvimento de um projeto. A análise estática pode encontrar problemas independentemente da entrada e da saída do programa, enquanto que a análise dinâmica pode encontrar problemas de codificação que não foram ainda informados como um padrão de erro nas ferramentas de análise estática. A primeira ferramenta de análise estática surgiu em meados dos anos 1970 (CHELF; CHOU, 2007), conhecida como Lint. Com o Lint, os programadores conseguiram pela primeira vez automatizar a checagem de código fonte em busca de erros.

Independente da linguagem de programação as ferramentas de análise estática podem fornecer alguns benefícios para o programador (CODEEXCELLENCE, 2012):

- Detecta áreas no código que precisam ser refatoradas ou simplificadas;
- Identifica áreas no código que podem precisar de mais testes ou de uma revisão mais aprofundada;

- Detecta problemas de *design* como Complexidade Ciclomática[1];
- Ajuda a reduzir a complexidade do código aumentando a manutenibilidade do software;
- Identifica potenciais problemas de qualidade no software antes do código ir para produção.

Os analisadores estáticos utilizam diversas técnicas para detectar partes do código problemáticas e que precisam ser melhoradas ou corrigidas. Existem quatro técnicas principais que são utilizadas pelas ferramentas (OWASP, 2017): análise de fluxo de dados, análise de fluxo de controle, *taint analysis* e análise léxica.

Os programas/algoritmos estão suscetíveis a diversas falhas causadas pelos programadores e que podem afetar o desenvolvimento dos mesmos em diversos pontos. Falhas como implementações ineficientes, problemas de segurança e trechos de código não usados degradam a qualidade final do produto. Quanto mais tempo um erro permanecer no projeto maior será o custo final do mesmo, pois à medida que o projeto/código cresce fica mais difícil encontrar e corrigir um erro. Erros que permanecem no código até a entrega do projeto ao usuário final podem causar danos ainda maiores ao projeto, como arquivos corrompidos, dados perdidos, instabilidade do serviço, etc.

As ferramentas de análise estática são capazes de detectar todos esses problemas que já foram citados sem ao menos colocar o programa em execução. A análise estática permite obter uma boa análise sobre problemas de codificação em pouco tempo (o tamanho do projeto pode influenciar o tempo necessário para rodar uma análise), possibilitando assim que a atividade de avaliar o código possa ser uma atividade periódica ao longo do desenvolvimento do projeto. Caso os problemas citados anteriormente sejam detectados logo nos estágios iniciais do desenvolvimento, o programador conseguirá entregar um programa de qualidade utilizando menos tempo e, consequentemente, menos dinheiro.

1.1 Motivação

Com a evolução da Computação, os softwares estão ficando cada vez mais complexos, com isso se torna cada vez mais necessário garantir a qualidade deles. A análise estática

[1] "Complexidade ciclomática é uma métrica do campo da engenharia de software, desenvolvida por Thomas J. McCabe em 1976, e serve para mensurar a complexidade de um determinado módulo (uma classe, um método, uma função etc), a partir da contagem do número de caminhos independentes que ele pode executar até o seu fim."(TEDESCO, 2017)

é uma técnica muito importante e que vem cada vez mais sendo utilizada para melhorar a qualidade dos programas, porém, até o momento, não existem muitos trabalhos que comparam as diversas ferramentas existentes no mercado.

Nesse contexto este trabalho foi realizado com a motivação de complementar e acrescentar os estudos comparativos já existentes sobre as ferramentas de análise estática. Nele, diversas ferramentas *open source* serão comparadas.

1.2 Objetivos

1.2.1 Gerais

Realizar um estudo comparativo entre as principais ferramentas de análise estática existentes no mercado para as linguagens C, Python e Java.

1.2.2 Específicos

Visando atingir os objetivos gerais desse trabalho, será necessário atingir alguns objetivos específicos, a saber:

- Definir os mecanismos de análise estática e seus objetivos;
- Comparar a análise estática à dinâmica;
- Entender como as ferramentas de análise estática funcionam;
- Definir um conjunto de erros comumente cometidos pelos programadores e que podem ser identificados por ferramentas de análise estática;
- Analisar a eficiência das ferramentas de análise estática em projetos reais;
- Relacionar a análise estática a etapas especificas do ciclo de desenvolvimento de software.

Esse trabalho está estruturado da seguinte maneira: O Capítulo 2 apresenta diversos conceitos importantes e que foram utilizados durante o desenvolvimento desse trabalho; O Capítulo 3 analisa e apresenta trabalhos relacionados; O Capítulo 4 descreve os experimentos que foram realizados e seus resultados; Por fim o Capítulo 5 apresenta as considerações finais desse trabalho.

2 Embasamento Teórico

Com o objetivo de compreender melhor como os analisadores estáticos funcionam, diversos artigos foram estudados durante o desenvolvimento desse trabalho. Nesse capítulo serão abordados os principais pontos que servirão como base para o desenvolvimento dessa monografia. O capítulo será dividido nas seguintes seções: Seção 1, introduz o processo de desenvolvimento de software; Seção 2, discute vulnerabilidades comuns a softwares; Seção 3, apresenta uma visão geral sobre teste de software; Seção 4, oferece uma visão geral da análise de programas (análise estática e dinâmica); Seção 5, apresenta classificações de analisadores estáticos.

2.1 Processo de desenvolvimento de *software*

Segundo o trabalho (GABRY, 2017) um processo de software é um conjunto de atividades relacionadas que conduzem a produção de um *software*. Essas atividades podem envolver o desenvolvimento de um *software* por completo ou a modificação de um sistema já existente. Segundo o autor, qualquer processo de *software* deve incluir essas quatro atividades:

1. Especificação do software: define as principais funcionalidades do sistema e as restrições relacionadas a ele;
2. *Design* do software e implementação;
3. Verificação e validação do software: garante que o software atende as especificações e as necessidades do cliente;
4. Evolução do software: o *software* sofre mudanças para atender as novas necessidades do cliente.

As ferramentas de análise estática são comumente aplicadas nas atividades de implementação e de validação do projeto. É recomendado que as ferramentas sejam aplicadas nos estágios iniciais da implementação, pois dessa forma a etapa de validação será menos

custosa, isso porque as ferramentas poderão encontrar *bugs*, problemas de segurança e de performance antes mesmo da fase de validação.

Geralmente o desenvolvimento ágil é o mais utilizado nas empresas de desenvolvimento de *software*, sendo que existem diversos métodos ágeis disponíveis como: *Scrum*, *Crystal*, *Agile Modeling* (AM), *Extreme Programming (XP)*, etc.

2.2 Falhas comuns no *software*

Falhas são *bugs*/problemas que afetam o uso do programa diante de alguma funcionalidade exigida ou comando específico realizado pelo usuário. Tais falhas também podem virar vulnerabilidades no software abrindo brechas de segurança que podem ser exploradas por *crackers*[1].

Nessa seção serão apresentadas as principais falhas encontradas em softwares.

2.2.1 *Buffer Overflow*

Uma situação de *Buffer overflow* ocorre quando um programa aloca mais dados que um *buffer*[2] está esperando receber, fazendo com que espaços adjacentes da memória sejam preenchidos com o excedente que foi passado para o *buffer*. Esse excedente pode permitir comportamentos inesperados, abrindo espaço para que que pessoas mal intencionadas executem qualquer conjunto de instruções no sistema operacional.

Esta vulnerabilidade pode ser evitada testando o tamanho dos dados que estão sendo passados para o *buffer*, fazendo com que o software possa bloquear dados que sejam maiores que o tamanho do *buffer*.

2.2.2 SQL *Injection*

"O SQL *Injection* é um ataque que visa enviar comandos nocivos à base de dados através dos campos de formulários ou através de URLs. Um ataque bem-sucedido pode, entre outras coisas, apagar (dropar) uma tabela do banco, deletar todos os dados da tabela

[1] "Os crackers são pessoas aficionadas por informática que utilizam seu grande conhecimento na área para quebrar códigos de segurança, senhas de acesso a redes e códigos de programas com fins criminosos"(MARTINS, 2012).

[2] "Os Buffers são áreas de memória criadas pelos programas para armazenar dados que estão sendo processados. Cada buffer tem um certo tamanho, dependendo do tipo e quantidade de dados que ele irá armazenar"(MORIMOTO, 2005).

ou até adquirir senhas que estejam cadastradas em um banco."(CARDOSO, 2010)

Essa vulnerabilidade permite que um código SQL qualquer seja injetado dentro de uma consulta SQL utilizada pelo sistema, por exemplo, caso essa vulnerabilidade existisse em uma consulta que é utilizada em um campo de pesquisa do projeto, o usuário poderia passar um comando SQL qualquer ao invés da consulta normal, dessa forma ele seria capaz de executar qualquer operação no banco de dados através desse campo.

2.2.3 Outras Falhas

Divisão por zero

Como o próprio nome sugere, esse tipo de erro ocorre quando o programa tenta fazer uma divisão por zero.

Ponteiro nulo

Um ponteiro é uma variável que é capaz de armazenar um endereço de memória ou assumir o valor nulo, que indica que não existe nenhum endereço válido associado ao ponteiro. O problema ocorre quando o programa tenta acessar alguma informação de um ponteiro que está nulo, fazendo com que comportamentos inesperados, violações de memória ou exceções sejam lançadas.

Vazamento de memória

Um vazamento de memória ocorre quando uma porção de memória é alocada e não é desalocada após o seu uso, fazendo com o que o programa ocupe uma certa quantidade de memória que não será mais utilizada para nada. Esses erros são bem comuns em C/C++ porém, geralmente não ocorrem em linguagens de programação modernas, estas linguagens contam com mecanismos que automaticamente limpam áreas de memória que não serão mais utilizadas.

***Overflow* de variável**

Toda variável possui um determinado tamanho (em *bytes*), esse tamanho é definido pelo tipo da variável e algumas vezes pelo conteúdo dela (no caso de strings, *arrays*, etc.), quando se tenta atribuir um valor com um tamanho maior do que é suportado pela variável podem acontecer comportamentos inesperados. Em Java e em C, por exemplo, parte do conteúdo pode ser perdido quando um valor com mais *bytes* é atribuído a uma variável com menos *bytes* de tamanho.

2.3 Teste de *Software*

Segundo WHITTAKER (WHITTAKER, 2000), teste de software é o processo de executar um sistema para determinar se ele atende as especificações e se é executado corretamente. De acordo com este autor, quando um usuário reporta um bug ao menos uma das situações se verifica:

- Ele executou um código que não foi testado;
- A ordem em que as instruções foram executadas diferem do que foi feito durante os testes;
- O usuário usou uma combinação de entradas não testada;
- O *software* nunca foi testado no ambiente que o usuário está usando.

É impossível garantir que um software está totalmente livre de *bugs* por meio de testes. O desenvolvedor ou testador de software portanto deve checar que este não apresente problemas para a maioria dos fluxos realizados pelo usuário. As técnicas apresentadas anteriormente podem e devem ser utilizadas para descobrir previamente *bugs* que o usuário pode encontrar durante a execução do programa.

As ferramentas de análise estática geralmente são usadas como um complemento às técnicas de teste de software, podendo encontrar *bugs* ou problemas de segurança que não foram encontrados nos testes.

2.4 Análise de programas

Essencialmente existem duas técnicas principais para fazer análise de programas, a análise estática e a análise dinâmica, as duas porém são técnicas bastante distintas e com propósitos diferentes. É recomendado que ambas sejam usadas em conjunto para que o código fonte seja melhor avaliado (GOMES et al., 2009).

A análise estática de software é realizada sem que o programa seja propriamente executado, sendo que algumas ferramentas analisam o código fonte enquanto outras analisam o código objeto do programa. Em outras palavras, analisadores estáticos analisam o "texto"do código sem tentar executá-lo (GOMES et al., 2009).

Dois conceitos bastante importantes na análise estática são os falsos positivos e os falsos negativos. O autor EMANUELSSON; NILSSON (EMANUELSSON; NILSSON, 2008) define falsos positivos como: problemas que são reportados pelas ferramentas mas que não são problemas de verdade. Por outro lado, os falsos negativos são definidos pelo autor como: problemas reais que não foram reportados pela ferramenta.

As ferramentas de análise estática são capazes de identificar diversos padrões de *bug*[3]:

- Códigos não utilizados (instruções, métodos, variáveis, classes);
- Uso errado de bibliotecas ou funções da linguagem;
- Códigos desnecessários, como estruturas de repetição que não necessariamente repetem (só é realizada uma iteração);
- Códigos duplicados na mesma classe ou em classes diferentes;
- Diversos *bugs* conhecidos de programação como: erro de compatibilidade entre operadores e operandos (no caso de Java podemos citar a comparação de *Strings* usando ==);
- Problemas de segurança conhecidos, como brechas no software que possibilitam o uso de *SQLInjection*.

O autor EMANUELSSON; NILSSON (EMANUELSSON; NILSSON, 2008) ainda cita alguns problemas adicionais que podem ser encontrados por ferramentas de análise estática:

- Instruções de alocação dinâmica de memória que não são sucedidas por instruções de desalocação;
- Operações ilegais como: divisão por zero, acesso de uma posição não existente do array, possíveis null pointers, etc.

Além desses problemas, o autor ainda cita que as ferramentas de análise estática podem ser usadas para a geração automática de casos de teste e para calcular métricas relacionadas ao software.

No início dos anos 1970, surgiu a primeira ferramenta de análise estática, conhecida por *Lint* (CHELF; CHOU, 2007). A criação desta ferramenta ocorreu logo após a revisão formal

[3] Alguns dos padrões de *bug* foram retirados da documentação oficial do *PMD*, disponível em `https://pmd.github.io/pmd-5.8.1/`

e as inspeções de código terem sido reconhecidas como peças chaves para a produtividade e a qualidade final do produto (GOMES et al., 2009). O *Lint* era um software que examinava códigos fontes escritos em C, era capaz de detectar erros e/ou melhorias no código que não eram encontrados normalmente por um compilador, como: variáveis e funções não usadas, linhas de código que não são alcançadas, uso de *longs* em *ints*. O *Lint* usava regras de tipo do C mais restritamente das usadas pelo compilador e era usado também para melhorar a compatibilidade do software final em diferentes máquinas e/ou sistemas operacionais (JOHNSON, 1977).

Por outro lado, a análise dinâmica é uma técnica que analisa e testa um programa em tempo de execução. Essa técnica é capaz de achar erros de lançamento de exceções indesejadas, *I/O*, memória, otimização, portabilidade, condições de corrida, etc.. Ao executar o software em todos os cenários para que ele foi designado, a análise dinâmica elimina a necessidade da criação artificial de situações para reproduzir erros (ROUSE, 2006). Alguns autores também chamam a análise dinâmica de *Runtime Error Detection*.

Apesar de sempre ser recomendado utilizar a análise estática juntamente com a análise dinâmica (GOMES et al., 2009), as duas são técnicas completamente distintas uma da outra e cada uma possui suas vantagens e desvantagens.

O autor CORNELL (CORNELL, 2008) e os autores GOMES et al. (GOMES et al., 2009) em seus trabalhos citam algumas vantagens e desvantagens da análise estática e da dinâmica, algumas delas podem ser encontradas na Tabelas 1 e 2.

Tabela 1 - Vantagens e desvantagens da análise estática.

Vantagens	**Desvantagens**
1. Não necessita gerar dados de entrada como acontece na análise dinâmica;	1. Necessita de acesso ao código fonte ou ao menos ao código objeto;
2. Já que a técnica tem acesso ao código fonte todos os possíveis comportamentos do programa são conhecidos;	2. Tipicamente necessita que o desenvolvedor proativamente rode a análise;
3. O código é levado a um estado para ser confiável, legível e com menor probabilidade de erros em testes futuros;	3. Não encontrará erros relacionados ao ambiente de *deployment* da aplicação, já que o código não é executado como na análise dinâmica;
4. Analisadores estáticos não precisam testar entrada e saída dos programas, podendo detectar bugs que seriam impossíveis de se detectar dinamicamente devido à alta quantidade de entradas possíveis para um programa.	4. A análise estática só faz sentido se aplicada juntamente com outras técnicas de revisão/detecção de problemas;
	5. Boas análises de código geralmente demandam algum tempo e muito processamento;
	6. Ferramentas de análise estática geralmente produzem uma grande quantidade de falsos positivos.

Tabela 2 - Vantagens e desvantagens da análise dinâmica.

Vantagens	**Desvantagens**
1. Só necessita do executável do programa, já que o teste é feito em tempo de execução;	1. Escopo limitado dos erros que podem ser encontrados, já que a qualidade da análise depende da quantidade e qualidade das entradas;
2. Ferramentas de análise dinâmica geralmente tem saídas bastante claras, o que requer um menor nível de conhecimento de quem está operando;	2. Não temos acesso às instruções que estão sendo executadas
3. Pode achar erros na infraestrutura, configuração ou outros erros relacionados ao ambiente de deployment.	

2.5 Classificação de analisadores estáticos

Em (NOVAK; KRAJNC et al., 2010) os autores criaram uma taxonomia para classificar ferramentas de análise estática. Essa taxonomia é bem completa e irá guiar a fase de experimentação desse trabalho. Os principais critérios de classificação serão descritos a seguir[4]:

- Tipo de Entrada - Código fonte ou código binário/objeto;
- Linguagens suportadas;
- Vulnerabilidades que são reportadas pelas ferramentas - Problemas de estilo, nomenclatura, problemas de concorrência, problemas de segurança e etc.;
- Extensibilidade - se a ferramenta pode ser melhorada com regras próprias;
- Tipo de licença - Código aberto, grátis ou comercial;

Com base nesses critérios, diversas ferramentas para as linguagens Java, Python e C foram classificadas. As ferramentas citadas aqui foram escolhidas com base no nosso

[4]Um conjunto completo com todas as classificações pode ser encontrado em (NOVAK; KRAJNC et al., 2010, p. 3)

conhecimento dessas ferramentas e também com base em uma lista de ferramentas encontrada no GitHub[5]. As ferramentas classificadas são apresentadas nas Tabelas 3, 4, e 5.

Tabela 3 - Classificação de ferramentas Java.

Ferramenta	Entrada	Linguagens	Regras	Extensível	Licença
PMD	Código Fonte	Java, Javascript, Apex, VisualForce, PLSQL, Apache Velocity, XML e XSL	Estilo, bugs, cálculo de complexidade, nomenclatura, performance, segurança e código não usado	Sim	Open Source
Checkstyle	Código Fonte	Java	Estilo, bugs, cálculo de complexidade e nomenclatura	Sim	Open Source
FindBugs	Código objeto	Java	Segurança, concorrência, performance, estilo, *bugs*, nomenclatura e código não usado	Sim	*Open Source*
Coverity	Código fonte	C/C++, Objective C, C#, Java, Javascript, PHP, Python, Node.js, Ruby, Fortran e Swift	Segurança, *bugs* e concorrência	Não	Comercial
SpotBugs	Código objeto	Java	Segurança, *bugs*, concorrência, estilo, nomenclatura, performance e código não usado	Sim	*Open Source*
Spoon	Código fonte	Java	Nenhuma (a ferramenta Spoon somente permite criar regras de análise estática ela não vem com nenhuma regra pré-definida)	Sim	*Open Source*

[5] A lista completa de ferramentas pode ser encontrada em: `https://github.com/mre/awesome-static-analysis`

Tabela 4 - Classificação de ferramentas Python.

Ferramenta	Entrada	Linguagens	Regras	Extensível	Licença
Bandit	Código fonte	Python	Segurança	Sim	*Open Source*
pycodestyle	Código fonte	Python	Estilo	Sim	*Open Source*
Pyflakes	Código fonte	Python	*Bugs*	Não	*Open Source*
Pylint	Código fonte	Python	Estilo, *bugs*, segurança e performance	Sim	*Open Source*
Python Taint	Código fonte	Python	Segurança	Sim	*Open Source*
PyChecker	Código fonte	Python	*Bugs* e estilo	Não	Grátis

Tabela 5 - Classificação de ferramentas C.

Ferramenta	Entrada	Linguagens	Regras	Extensível	Licença
CodeSonar	Código fonte e compilado	C/C++	Segurança, *bugs* e concorrência	Não	Comercial
Cppcheck	Código fonte	C/C++	*Bugs* e código não usado	Sim	*Open Source*
Flawfinder	Código fonte	C/C++	Segurança	Não	*Open Source*
OCLint	Código fonte	C/C++	*Bugs*, código não usado, cálculo de complexidade, estilo	Sim	*Open Source*
PolySpace Bug Finder	Código fonte	C/C++	*Bugs*, nomenclatura, cálculo de complexidade, estilo e segurança	Não	Comercial
PolySpace Code Prover	Código fonte	C/C++	*Bugs*	Não	Comercial
Splint	Código fonte	C	*Bugs*, segurança e nomenclatura	Sim	*Open Source*

Apesar de a utilização das ferramentas de análise estática ser bem simples, elas envolvem bastantes conceitos necessários a sua melhor compreensão. Esse capítulo introduziu os conceitos necessários ao desenvolvimento dessa monografia. No próximo capítulo serão apresentados os trabalhos relacionados e como eles se comparam à este trabalho.

3 Trabalhos relacionados

Durante a escrita desse trabalho foram escolhidos diversos artigos com propostas parecidas à desenvolvida aqui, com o objetivo de apresentar o estado da arte da área. Tais artigos introduzem a análise estática e/ou fazem comparativos e análises em ferramentas associadas. Estes artigos foram encontrados no *Google Scholar*[1], utilizando a palavra chave *static analysis* e selecionando aqueles com um grande número de citações. Outros trabalhos foram selecionados com base nas referências dos artigos previamente selecionados.

3.1 An overview on the Static Code Analysis approach in Software Development

Em (GOMES et al., 2009), são mostradas as principais ferramentas de análise estática existentes para as linguagens .NET, C/C++ e Java. Os autores apresentam aproximadamente 20 ferramentas explicando o funcionamento delas. Para a linguagem .NET são apresentadas as ferramentas *FxCop*, *StyleCop* e *CodeIt.Right*; para Java são apresentadas as ferramentas *FindBugs*, *PMD*, *CheckStyle*, *JLint* e *ESC/Java*; e para C/C++ são apresentadas as ferramentas *Lint*, *CodeSonar*, *HP Code Advisor*, *Mygcc*, *Splint* e *PolySpace Verifier*. O artigo também apresenta algumas ferramentas que servem para diversas linguagens simultaneamente.

Para comparar as ferramentas abordadas no artigo, os autores as aplicam sobre exemplos de código-fonte de aplicações conhecidas: SendMail (SM), BIND e WU-FTPD. Esses exemplos de código continham 14 ocorrências diferentes da vulnerabilidade de *buffer overflow*.

Na primeira rodada dos testes, foram comparadas as ferramentas *ARCHER*, *BOON*, *PolySpace C Verifier*, *Splint*, e *UNO tools*. Nesses testes, a ferramenta *Archer* detectou somente uma vulnerabilidade, a *UNO* não detectou nenhuma, a *Boon* gerou dois falsos positivos de *buffer overflow* e as outras duas ferramentas - *Splint* e *PolySpace* - detectaram boa parte das vulnerabilidades.

[1]Disponível em `https://scholar.google.com.br/`.

Na segunda rodada foram usadas as ferramentas *PolySpace Verifier*, *Coverity Prevent* e *Klocwork K7*, porém as ferramentas foram comparadas quanto às funcionalidades oferecidas, ao invés de serem compararadas utilizando *benchmarks* como foi feito na rodada um. Os autores citam que a ferramenta *Coverity* e a *Klocwork* sacrificam a possibilidade de encontrar todos os bugs para diminuir a quantidade de falsos positivos, enquanto que a ferramenta *PolySpace* produz uma quantidade significativamente maior de falsos positivos mas é capaz de encontrar mais problemas que as outras duas.

Com o resultado dos testes, os autores indicam quais ferramentas se saíram melhor em detectar falhas de segurança. Por fim concluiu-se que diante de uma alta taxa de falso positivos, as análises realmente úteis podem acabar sendo ignoradas pelos programadores.

O trabalho foi uma boa contribuição para o desenvolvimento dessa monografia, pois diversos conceitos importantes foram bem explicados e muitas ferramentas foram introduzidas. Porém, os autores poderiam ter sido mais profundos nas comparações que foram feitas no artigo, pois as ferramentas só foram expostas a um tipo de problema de segurança e elas não foram comparadas diante de projetos reais. Para uma comparação mais fiel e confiável, as ferramentas deveriam ter sido expostas a mais problemas de codificação e falhas de segurança, com o objetivo de avaliar a eficácia delas.

Os autores utilizaram uma proposta bem parecida à que vai ser desenvolvida aqui. Grande parte das ferramentas para Java e C introduzidas pelos autores serão também consideradas para esse trabalho, assim como algumas técnicas utilizadas para comparar as ferramentas. Nessa monografia o foco é a fase de experimentação, com mais comparações e com mais casos de teste, provendo assim uma maior base para avaliar as diversas ferramentas existentes.

3.2 A Comparative Study of Industrial Static Analysis Tools

No trabalho (EMANUELSSON; NILSSON, 2008) são comparadas três ferramentas pagas: a *Klockwork K7*, *Coverity Prevent* e *PolySpace*. Inicialmente, os autores comparam as funcionalidades das ferramentas e indicam que as ferramentas *Coverity* e *Klockwork* sacrificam a identificação de todos os erros para apresentar uma menor quantidade de falsos positivos, enquanto que a *PolySpace* tenta mostrar todos os erros, consequentemente apresentando mais falsos positivos.

Os autores também comparam o tempo que as ferramentas demoram para realizar

as análises e se as análises são progressivas ou não. Análises progressivas não precisam analisar todo o código quando alguma parte deste é mudada, dessa forma a ferramenta só irá analisar os blocos que foram mudados, diminuindo drasticamente o tempo necessário para finalizar a análise. Das três ferramentas, somente a *Coverity* suporta análise progressiva. Quanto ao tempo, os autores citam que a *PolySpace* não é capaz de analisar bases de código muito grandes (acima de 50 mil linhas de código) em um tempo aceitável.

Para comparar as ferramentas, os autores as submeteram a alguns projetos da empresa *Ericsson* e colheram algumas opiniões do programadores que trabalhavam lá (os autores citam que o time da *Ericsson* não foi capaz de avaliar a ferramenta *PolySpace*, pois esta não retornou resultados significativos, mesmo rodando por dias). Com os erros reportados pelas ferramentas, os autores compararam a quantidade de falsos positivos, falsos negativos e quantos erros reais as ferramentas foram capazes de detectar. Em algumas avaliações também foram usados projetos com erros já conhecidos e detectados para avaliar quantos deles iriam ser apontados por cada uma das ferramentas. Algumas das conclusões apresentadas no artigo após a experimentação das ferramentas *Klocwork* e *Coverity* são apresentadas a seguir[2]:

- As ferramentas são fáceis de instalar;
- As ferramentas foram capazes de encontrar bugs que seriam difíceis de encontrar com outra técnica;
- O tempo gasto para analisar um código fonte é praticamente o mesmo de rodar uma *build* (mesmo para projetos com milhões de linhas de código);
- A quantidade de falsos positivos é aceitável, mesmo em aplicações grandes;
- Muitos usuários esperavam que a ferramenta detectasse mais erros e erros que fossem mais severos, porém muitos usuários também se impressionaram com o fato das ferramentas terem detectado erros mesmo em aplicações bastante testadas.

Os autores também submeteram as ferramentas a um código antigo onde diversas falhas tinham sido corrigidas utilizando testes. O objetivo dessa avaliação era saber quantos erros seriam detectados pelas ferramentas de análise estática e que também foram detectados pelos testes. Os autores acabam não apresentando qual a proporção entre os erros apontados pelas ferramentas e os que foram identificados usando testes.

[2]Um conjunto completo com todas as conclusões pode ser encontrado em (EMANUELSSON; NILSSON, 2008, p. 15)

Esse trabalho (EMANUELSSON; NILSSON, 2008) foi uma base muito boa para esta monografia, pois a comparação feita pelos autores é profunda e avalia diversos pontos. Porém, o artigo foi escrito em 2008 e possivelmente muito dos pontos apontados pelos autores já não são mais verdade hoje, o trabalho compara poucas ferramentas (somente 3). Nessa monografia iremos cobrir uma quantidade maior de ferramentas e as análises serão feitas sobre projetos reais (assim como o autores fizeram com os projetos da *Ericsson*).

3.3 Test-driving static analysis tools in search of C code vulnerabilities

No artigo (CHATZIELEFTHERIOU; KATSAROS, 2011) os autores apresentam 6 ferramentas de análise estática para a linguagem C (sendo 4 *open source* e 2 comerciais) e as comparam considerando a quantidade de erros que elas são capazes de encontrar. Nos experimentos também foi levado em consideração a quantidade de memória e o tempo gasto na análise.

O artigo realizou os experimentos com as ferramentas apresentadas abaixo na experimentação:

- Splint
- UNO
- Cppcheck
- Frama-C
- "Comercial B"
- Parasoft C++ Test

Destas, apenas as duas últimas são ferramentas pagas, sendo as demais *open source*. Por motivos legais os autores não citam o nome da segunda ferramenta comercial avaliada, no trabalho ela é citada como "Comercial B".

No artigo as ferramentas são comparadas utilizando um suíte de teste, com programas que contém entre 1000 e 7000 linhas de código, com diversos defeitos de segurança e de erros de programação. A suíte inclui 30 defeitos de código distintos selecionados do catálogo *Common Weakness Enumeration* (CWE) e do padrão *CERT C Secure Coding*.

Alguns dos defeitos utilizados no artigo podem ser encontrados abaixo[3]:

- Divisão de valor por zero;
- Variáveis não inicializadas;
- Variável com um valor maior do que é suportado pela representação interna;
- Acesso a posições inválidas de um *array*;
- Acesso a posições inválidas de uma *string*;
- Chamada duplicada da instrução *free()* em um endereço de memória;
- Memória utilizada que não foi desalocada.

Os autores quantificam a eficiência das ferramentas usando cinco métricas: exatidão, precisão, revocação, especificidade e *F-measure.*

A exatidão é a relação entre as quantidade de erros reportados sobre o número de erros presentes na suíte de testes. A precisão é a relação da quantidade de erros reportados que são realmente erros (positivos verdadeiros). A métrica que eles chamam de revocação é a relação do número de positivos verdadeiros sobre o número da quantidade de erros presentes na suíte. Especifidade é a relação da quantidade de negativos verdadeiros[4] sobre a soma de negativos verdadeiros com os falso positivos. Por fim o *F-measure* é uma métrica agregada das métricas de precisão e revocação.

Para finalizar as experimentações, os autores comparam o tempo médio e a quantidade de memória usada pelas ferramentas para analisar a suíte de testes de acordo com a quantidade de linhas do programa analisado.

Como resultado da experimentação os autores concluíram que:

- A ferramenta "Comercial B"(nome não revelado pelo autor por motivos legais) possui a melhor exatidão e a maior *F-Measure*;
- A ferramenta *Frama C* possui a melhor precisão e a melhor especificidade;

[3] Um conjunto completo com todos os defeitos pode ser encontrado em (CHATZIELEFTHERIOU; KATSAROS, 2011, p. 97)

[4] Um negativo verdadeiro ocorre quando uma instrução é colocada no teste na intenção de confundir a ferramenta de análise, se a ferramenta não reportar essa instrução como erro ela é considerada um negativo verdadeiro.

- A ferramenta *C++ Test* possui a melhor revocação;

Em termos de velocidade, a ferramenta *Frama-C* foi a mais lenta, demorando cerca de 35 segundos para analisar um programa com 7000 linhas de código, enquanto que a ferramenta mais rápida foi a *UNO*, que demorou algumas centenas de milissegundos para analisar a mesma quantidade de linhas.

Sobre o uso de memória, a ferramenta *Splint* foi a que usou mais memória chegando a consumir um pouco mais de 180Mb para um programa com 7000 linhas. A ferramenta *UNO* foi a que consumiu menos, utilizando menos que 10Mb para a mesma quantidade de linhas de código

O artigo utiliza uma abordagem bem interessante para experimentar as ferramentas, a quantidade de erros utilizadas na experimentação é muito boa e reflete bem a realidade, já que analisa os defeitos mais cometidos pelos programadores. Ao comparar o tempo e memória utilizada durante a análise, os autores permitem que o leitor possa avaliar se vale a pena trocar a eficiência da ferramenta pela eficácia.

Por fim, o artigo é limitado ao tamanho da suíte de testes criada pelos autores e não considera projetos reais (somente simulações de vários erros de programação reais), além disso o artigo foi escrito em 2011 e possivelmente boa parte do que foi experimentado está desatualizado. Nessa monografia as ferramentas serão comparadas utilizando 30 programas reais (10 para C, 10 para Python e 10 para Java) o que deixa a experimentação mais abrangente e diversificada.

3.4 Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools

No trabalho (KRATKIEWICZ; LIPPMANN, 2005) são apresentadas 5 ferramentas de análise estática, estas são comparadas em relação à eficiência em detectar vulnerabilidades explorando a técnica de *buffer overflow*. Os autores aplicam as ferramentas a um conjunto de 291 pequenos programas C com diferentes ocorrências de *buffer overflows*.

O artigo utilizou uma taxonomia com 22 atributos[5] que classifica as formas em que as vulnerabilidades de *buffer overflow* podem aparecer. Essa taxonomia foi utilizada para

[5] Um conjunto completo com todos os atributos pode ser encontrado em (KRATKIEWICZ; LIPPMANN, 2005, p. 2)

guiar a criação dos casos de teste. Com a taxonomia os autores conseguiram cobrir uma maior variedade de defeitos de *buffer overflow*.

No artigo foram produzidos, pelos autores, 291 programas em C, cada programa com quatro versões: sem *buffer overlow*, com *overlows* mínimos (*overflow* de um byte), com *overflows* médios (*overflow* de oito bytes) e com *overflows* grandes (*overflow* de 4096 bytes).

As ferramentas utilizadas pelos autores na fase de experimentação foram: *ARCHER*, *BOON*, *PolySpace*, *Splint* e *UNO*, todas para a linguagem de programação C. As comparações foram feitas utilizando 3 métricas:

- Taxa de detecção: quantidade de erros detectados a partir da quantidade de testes avaliados;
- Taxa de falsos positivos: quantidade de falsos positivos a partir da quantidade de testes avaliados;
- Taxa de confusão: quantidade de testes onde a ferramenta reportou erro tanto na versão sem *overflow* (falso positivo) quanto na versão com *overflow* a partir da quantidade de erros detectados corretamente.

Os autores também apresentam os tempos de execução das cinco ferramentas de análise estática utilizadas para analisar os códigos.

Segundo os autores, a ferramenta *PolySpace* foi a que apresentou a maior taxa de detecção (99,7%), enquanto que a BOON foi a que apresentou a menor (0,7%). As ferramentas *ARCHER*, *BOON* e *UNO* não apresentaram nenhum falso positivo, enquanto que a *PolySpace* apresentou 2,4% de falsos positivos e a *Splint* 12%. Quanto a taxa de confusão, as ferramentas *ARCHER*, *BOON*, e *UNO* novamente ficaram com 0%, enquanto que a *PolySpace* ficou com 2,4% e a *Splint* com 21,3%.

O grande destaque da experimentação foi a ferramenta *PolySpace*, que somente deixou de detectar um erro e gerou somente três falsos positivos. Porém, por ter uma elevada taxa de detecção, a ferramenta *PolySpace* demorou muito para executar, levando um tempo quase 2000 vezes maior do que a média das outras ferramentas: enquanto a *PolySpace* levou 56 horas para finalizar as análises, as outras levaram em média 103 segundos para fazer as mesmas análises.

A proposta do artigo é bem interessante, a experimentação foi feita com bastante

cuidado e a taxonomia garantiu que uma grande quantidade de versões de códigos com diferentes ocorrências de *buffer overflow* fossem testadas.

O artigo somente analisa as ferramentas na detecção de *buffer overflows* na linguagem C e, como nos outros trabalhos relacionados, a experimentação é realizada somente sobre programas fictícios simulando erros reais. Nessa monografia serão analisados 30 programas reais (10 para cada linguagem) e os erros analisados não são limitados a um tipo de vulnerabilidade ou a uma linguagem de programação, são limitados portanto pelos erros presentes nos projetos que serão analisados.

3.5 Um estudo sobre a correlação entre defeitos de campo e *warnings* reportados por uma ferramenta de análise estática

No trabalho (FILHO et al., 2010), os autores propõe avaliar se as ferramentas de análise estática são eficazes em detectar defeitos que foram reportados por usuários dos *softwares*. Nessa avaliação foi utilizada a ferramenta *FindBugs* para a linguagem Java. O artigo propõe responder duas perguntas: se as ferramentas de análise estática ajudam a remover defeitos reportados por usuários finais e se os padrões de bugs detectados pelas ferramentas podem indicar defeitos que serão reportados pelos usuários posteriormente.

Para responder a primeira pergunta os autores realizaram análises utilizando a ferramenta *FindBugs* e 2 projetos armazenados no repositório *iBugs*[6], o *AspectJ* e o *Rhino*. No *iBugs* são disponibilizados sistemas para realizar *benchmarks* sobre ferramentas de análise estática. Com esses 2 projetos os autores coletaram um histórico de *bugs* e uma versão do *software* com o bug e uma versão onde o *bug* tinha sido corrigido.

Ao final da primeira avaliação, os autores concluíram que as ferramentas de análise estática não ofereceram muita ajuda aos desenvolvedores em detectar problemas de *software*, os quais chegaram até o usuário final: para o projeto *Rhino*, a ferramenta FindBugs só encontrou padrões de *bugs* em 58% das versões, enquanto que no projeto *AspectJ* apenas em 18%. No entanto, os autores afirmam que para uma conclusão mais segura seriam necessários novos estudos abordando uma quantidade maior de casos.

Para responder a segunda pergunta, foram analisados 25 *softwares* da fundação *Apache* utilizando a ferramenta *FindBugs*. Nessa análise, foram realizadas duas execuções da

[6] O *iBugs* pode ser acessado através de: `https://www.st.cs.uni-saarland.de/ibugs/`

ferramenta sobre os projetos. Na primeira execução, foi utilizada a configuração padrão da ferramenta, enquanto que na segunda a ferramenta foi configurada somente para mostrar os erros mais prioritários. Para cada execução, a quantidade de padrões de *bugs* reportados foi calculada e utilizada para comparar com a quantidade de defeitos que foram reportados nas ferramentas. Para comparar essas duas medidas, foi utilizado o teste de correlação de *ranks* de Spearman que permite identificar se existe alguma correlação sobre os dados.

Ao final dessa segunda avaliação, foi concluído que existe uma grande correlação estatística entre a quantidade de padrões de *bugs* encontrados pela ferramenta e os defeitos que foram reportados por usuários finais. Além disso, os autores sugerem que se a ferramenta *FindBugs* for configurada para reportar somente os erros de maior prioridade, o programador conseguirá facilmente analisar todos os padrões de *bugs* reportados.

O artigo tem uma proposta bastante interessante, pois a experimentação realizada difere das realizadas nos outros artigos. Neste artigo, os autores foram mais a fundo do que simplesmente avaliar *benchmarks* sobre as ferramentas, eles mostraram através de dados estatísticos se as ferramentas são realmente eficazes em detectar defeitos no código. Porém, o artigo só realiza o estudo sobre uma ferramenta e em uma linguagem, e sabemos que os resultados alcançados podem diferir para outras ferramentas ou linguagens. Nessa monografia, analisamos 14 ferramentas para 3 linguagens diferentes (Java, Python e C), permitindo que os resultados não sejam limitados a só uma ferramenta ou uma linguagem.

3.6 Considerações acerca da análise inicial

Os trabalhos analisados aqui foram de grande importância para o desenvolvimento dessa monografia, pois apresentaram técnicas que poderiam ser utilizadas para guiar o estudo comparativo desse trabalho. Existem diversos artigos publicados com o objetivo de comparar e estudar ferramentas de análise estática, porém a maioria desses artigos só realizam a comparação utilizando conceitos, técnicas e erros comumente cometidos pelos programadores, eles não analisam as ferramentas utilizando projetos reais. Uma comparação entre os artigos pode ser encontrada na Tabela 6.

Tabela 6 - Comparação entre os trabalhos relacionados.

Artigo	**Linguagens cobertas**	**Ferramentas utilizadas**
An overview on the Static Code Analysis approach in Software Development (GOMES et al., 2009)	C	ARCHER, BOON, PolySpace, Splint, UNO Tools, Coverity Prevent e KlockWork K7
A Comparative Study of Industrial Static Analysis Tools (EMANUELSSON; NILSSON, 2008)	C	Klockwork K7, Coverity Preven e PolySpace
Test-driving static analysis tools in search of C code vulnerabilities (CHATZIELEFTHERIOU; KATSAROS, 2011)	C	Splint, UNO, Cppcheck, Frama-C e Parasoft C++ Test
Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools (KRATKIEWICZ; LIPPMANN, 2005)	C	ARCHER, BOON, PolySpace e UNO
Um estudo sobre a correlação entre defeitos de campo e *warnings* reportados por uma ferramenta de análise estática (FILHO et al., 2010)	Java	FindBugs

Com exceção do (EMANUELSSON; NILSSON, 2008), ao qual aplicou os experimentos na empresa *Ericsson*, os critérios de comparação encontrados nos artigos não levam em consideração projetos reais que serão utilizados pelos programadores no uso diário das ferramentas de análise estática.

Nessa monografia o estudo comparativo também será baseado em projetos reais encontrados no *GitHub* (além de dois projetos tirados da empresa *Simbiose Ventures*) o que traz resultados mais próximos à realidade dos programadores e ou empresas de desenvolvimento de software.

4 Estudo Comparativo

Nesse capítulo será definido o cenário de estudo comparativo que foi utilizado nesse trabalho. Na Seção 1 são definidos os objetivos do estudo. Na Seção 2 é descrito o ambiente onde o estudo foi realizado. Na Seção 3 e 4 o primeiro e o segundo estudo são analisados, respectivamente. Finalmente, na seção 5 é apresentada a conclusão geral sobre os estudos.

4.1 Objetivos

A fase de estudo comparativo dessa monografia tem como objetivo principal comparar diversas ferramentas de análise estática. Nessa fase foram comparadas ferramentas *open source* para três linguagens diferentes: Java, Python e C. Nessa etapa foram realizados dois estudos.

No primeiro estudo, as ferramentas foram comparadas utilizando diversos *benchmarks* baseados nos utilizados no artigo *"Test-driving static analysis tools in search of C code vulnerabilities"* (CHATZIELEFTHERIOU; KATSAROS, 2011), esse estudo teve como objetivo comparar os resultados obtidos pelos autores com os obtidos pelas ferramentas que foram escolhidas para essa monografia, além disso o estudo deu uma nova visão sobre os resultados dos *benchmarks*, já que o artigo foi escrito em 2011 e certamente as ferramentas evoluíram de lá pra cá.

Já no segundo estudo, as ferramentas foram submetidas a trinta projetos (dez para cada linguagem) do *GitHub* com o objetivo de comparar a eficiência de tempo e de memória das ferramentas em realizar as análises. Nesse estudo foram comparados o tempo de execução, o uso de memória e a quantidade de erros apresentados. Além disso, as ferramentas foram avaliadas com base na interface com o usuário e no nível de detalhamento dos erros apresentados.

4.2 Ambiente do Estudo Comparativo

Os dois estudos foram realizados em uma máquina rodando Linux 4.9.x com as seguintes configurações:

- Processador *Intel Core i7-4500U* com 2 núcleos, 4 *threads*, frequência base de 1,8 Ghz e frequência máxima de 3,0 Ghz;
- 8Gb de Memória *RAM* DDR3 com 1800Mhz.

Versão das linguagens utilizadas:

- Java: OpenJDK 1.8.0_144
- Python: 3.6.2
- C: gcc 7.2.0

4.3 Estudo Comparativo 1

No primeiro estudo comparativo as ferramentas de análise estática foram submetidas a um conjunto de *bugs* retirado do catálogo CWE (*Common Weakness Enumeration*)[1] para analisar a eficácia delas em encontrar os *bugs*. O mesmo conjunto de *bugs* que foi utilizado no artigo *"Test-driving static analysis tools in search of C code vulnerabilities"* (CHATZIELEFTHERIOU; KATSAROS, 2011) será utilizado aqui[2]. O conjunto de regras que foi utilizado pode ser encontrado na Tabela 7.

Para esse estudo, foram desenvolvidos pequenos programas que possuíam bugs no código fonte. Para cada linguagem e *bug* foi criada uma implementação diferente. Os códigos fonte para os *benchmarks* utilizados foram disponibilizados no GitHub[3].

[1] O CWE pode ser acessado nesse link `https://cwe.mitre.org/`.

[2] Os *bugs* CWE-367, CWE-665 e CWE-691 não foram abordados nessa monografia pois faltavam instruções de como escrever os programas para realizar os *benchmarks*.

[3] O código fonte completo pode ser encontrado em: `https://github.com/joaomedeiros95/static-analysis-tools-experiments`

Tabela 7 - Conjunto de regras utilizado no estudo comparativo 1.

Código	**Descrição**
CWE-119	Acesso a posições fora do array
CWE-120	*Buffer overflow* com array *Buffer overflow* com string
CWE-134	Falha de segurança na formatação de strings
CWE-170	Strings sem a indicação de fim
CWE-190	*Overflow* de varíavel
CWE-193	Acesso de uma posição a mais ou a menos em um array Acesso de uma posição a mais ou a menos em uma string
CWE-195	Conversão de variável com sinal para variável sem sinal
CWE-197	Perda de precisão numérica
CWE-222	Perda de informação em strings
CWE-369	Divisão por zero
CWE-401	Vazamento de memória com malloc Vazamento de memória com calloc Vazamento de memória com realloc
CWE-403	Arquivos não fechados após o uso
CWE-415	Chamada dupla à instrução de desalocação (*free*)
CWE-416	Uso de uma posição de memória que já foi liberada
CWE-457	Variável não inicializada (int) Variável não inicializada (ponteiro) Variável não inicializada (*array*)
CWE-476	Null pointer
CWE-667	Instrução de *lock* que não é seguida de uma instrução de *unlock*

Além dos *bugs* citados na Tabela 7, alguns bugs utilizados no artigo (CHATZIELEFTHERIOU; KATSAROS, 2011) também fizeram parte do estudo:

- Acesso a um arquivo que já foi fechado;
- Chamada de uma instrução inválida para o modo do arquivo;
- Chamada dupla de instrução para fechar um arquivo;
- Acesso a um arquivo que não foi aberto;
- Falta de checagem de possível falha ao abrir arquivos.

Para esse estudo foram utilizadas todas as ferramentas não comerciais citadas na Seção 2.5 dessa monografia.

4.3.1 Resultados do Estudo Comparativo

Esta seção apresenta os resultados obtidos no estudo, separados por linguagem. Para as linguagens Python e Java alguns *benchmarks* não foram realizados pois estes eram específicos da linguagem C.

Linguagem C

Os resultados para as ferramentas C podem ser encontrados na Tabela 8, o símbolo ✔ indica que o *bug* foi detectado corretamente, por outro lado o símbolo ✖ indica que o *bug* não foi identificado.

Tabela 8 - Resultados do estudo 1 para as ferramentas C.

Bugs	**Cppcheck**	**FlawFinder**	**OCLint**	**Splint**
Divisão por zero (explícita)	✔	✖	✔	✖
Divisão por zero (implícita)	✔	✖	✖	✖
Null pointer	✔	✖	✖	✔
Variável não inicializada (inteiro)	✔	✖	✖	✔
Variável não inicializada (ponteiro)	✔	✖	✖	✔
Variável não inicializada (array)	✖	✖	✖	✔
Overflow de inteiro direto	✖	✖	✔	✖
Overflow de inteiro com soma	✔	✖	✖	✖
Conversão de variável com sinal para variável sem sinal	✖	✖	✖	✔
Perda de precisão numérica	✖	✖	✖	✔
Acesso à posições fora do array	✔	✖	✔	✖
Acesso de uma posição a mais ou a menos em um array	✔	✖	✖	✖
Buffer overflow com array	✖	✔	✖	✖
Strings sem a indicação de fim	✖	✖	✖	✖
Acesso de uma posição a mais ou a menos em uma string	✔	✖	✖	✖
Perda de informação em strings	✖	✖	✖	✖
Buffer overflow com string(strcpy)	✔	✔	✖	✖
Buffer overflow com string(strcat)	✔	✔	✖	✖
Buffer overflow com string(gets)	✔	✔	✖	✔
Buffer overflow com string(sprintf)	✔	✔	✖	✔
Buffer overflow com string(strncpy)	✔	✔	✖	✖
Buffer overflow com string(strncat)	✔	✔	✖	✖
Buffer overflow com string(fgets)	✔	✖	✖	✖
Buffer overflow com string(snprintf)	✔	✖	✖	✖
Falha de segurança na formatação de strings	✖	✔	✖	✔

Chamada dupla em escopos diferentes à instrução de desalocação	✔	✖	✖	✖
Chamada dupla no mesmo escopo à instrução de desalocação	✔	✖	✖	✔
Vazamento de memória com malloc	✔	✖	✖	✔
Vazamento de memória com calloc	✔	✖	✖	✔
Vazamento de memória com realloc	✔	✖	✖	✖
Uso de uma posição de memória que já foi liberada	✖	✖	✖	✔
Acesso a um arquivo que já foi fechado	✔	✖	✖	✖
Chamada de uma instrução inválida para o modo do arquivo	✔	✖	✖	✖
Chamada dupla de instrução para fechar arquivo	✔	✖	✖	✖
Arquivos não fechados após o uso	✔	✖	✖	✖
Acesso a um arquivo que não foi aberto	✔	✖	✖	✔
Falta de checagem de possível falha ao abrir arquivos	✖	✖	✖	✔
Instrução de *lock* que não é seguida de uma instrução de *unlock*	✖	✖	✖	✖

Tanto nessa monografia quanto em (CHATZIELEFTHERIOU; KATSAROS, 2011) foram utilizadas duas ferramentas iguais, a *Cppcheck* e a *Splint*. A maioria dos resultados foram exatamente iguais aos do artigo porém em alguns dos *benchmarks* houve divergências. As divergências dos resultados dos *benchmarks* realizados pelos autores e os *benchmarks* dessa monografia podem ser encontrados na Tabela 9. Tais divergências podem ter acontecido por dois motivos: os resultados do artigo estão desatualizados já que ele foi escrito a sete anos atrás ou pelo fato dos códigos fontes que foram escritos nessa nessa monografia diferirem dos do artigo[4].

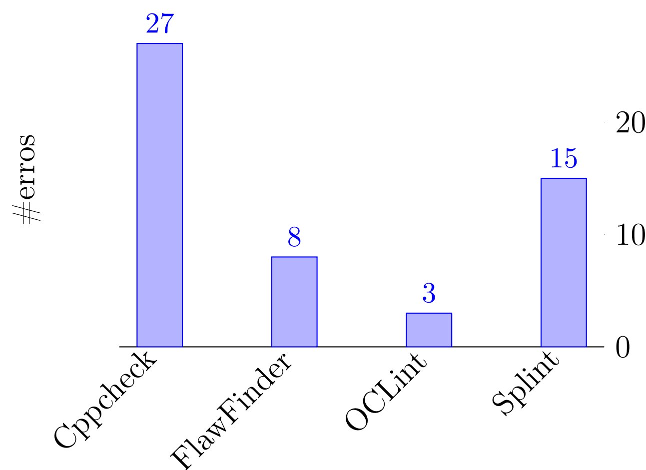
[4]Não foi possível analisar o código os *benchmarks* usado pelos autores, pois o link disponibilizado no artigo está fora do ar.

Tabela 9 - Diferenças entre os resultados do artigo e os obtidos nessa monografia.

Bug	**Artigo**	**Monografia**
Null pointer	Ferramenta *Cppcheck* não detectou.	Ferramenta *Cppcheck* detectou.
Variáveis não inicializadas	Ferramenta *Cppcheck* não detectou nenhum dos casos.	Ferramenta *Cppcheck* somente não detectou arrays não inicializados.
Overflow de inteiro	Nenhuma das ferramentas detectou o problema.	Ferramenta *OCLint* detectou o *overflow* direto e a *Cppcheck* detectou o *overflow* na soma.
Perda de precisão numérica	Ferramenta *Splint* não detectou.	Ferramenta *Splint* detectou.
Buffer overflow com array	Nenhuma ferramenta detectou.	Ferramenta *Flawfinder* detectou.
Uso de uma posição de memória que já foi liberada	Ferramenta *Cppcheck* detectou.	Ferramenta *Cppcheck* não detectou.
Acesso a um arquivo que já foi fechado	Ferramenta *Cppcheck* não detectou.	Ferramenta *Cppcheck* detectou.
Chamada de uma instrução inválida para o modo do arquivo	Nenhuma ferramenta detectou.	Ferramenta *Cppcheck* detectou.
Acesso a um arquivo que não foi aberto	Nenhuma ferramenta detectou.	Ferramentas *Cppcheck* e *Splint* detectaram.

A ferramenta *Cppcheck* foi a que detectou mais *bugs* dentro as quatro ferramentas avaliadas. Na Figura 1 é apresentada a quantidade de erros detectados por cada ferramenta.

Figura 1 - Quantidade de padrões de bugs detectados nas ferramentas C.



Linguagem Java

Na linguagem Java alguns *benchmarks* não puderam ser compilados pela máquina virtual Java, pois estes foram detectados pelo compilador como bug, portanto tais *benchmarks* não puderam ser executados nas ferramentas *FindBugs* e *SpotBugs* que analisam o código objeto do programa. Esses casos foram marcados com o símbolo ➖.

Os resultados para as ferramentas Java se encontram na Tabela 10.

Tabela 10 - Resultados do estudo comparativo 1 para as ferramentas Java.

Bug	**PMD**	**Checkstyle**	**FindBugs**	**SpotBugs**
Divisão por zero	✖	✖	✖	✖
Null pointer	✖	✖	✔	✔
Variável não inicializada (inteiro)	✔	✖	➖	➖
Variável não inicializada (objeto)	✔	✖	➖	➖
Variável não inicializada (array)	✔	✖	➖	➖
Overflow de inteiro direto	✖	✖	➖	➖
Overflow de inteiro com soma	✖	✖	✖	✖
Perda de precisão númerica	✖	✖	✖	✖
Acesso à posições fora do array	✖	✖	✔	✔
Acesso de uma posição a mais ou a menos em um array	✖	✖	✖	✖
Acesso à um arquivo que já foi fechado	✖	✖	✖	✖
Chamada dupla de instrução para fechar arquivo	✖	✖	✖	✖
Arquivos não fechados após o uso	✖	✖	✔	✔
Falta de checagem de falha ao abrir arquivos	✖	✖	✖	✖
Instrução de *lock* que não é seguida de uma instrução de *unlock*	✖	✖	✖	✖

Devido o foco maior da ferramenta *Checkstyle* ser o estilo de código, ela não detectou nenhum dos *bugs* nos *benchmarks*. Se os *benchmarks* tivessem problemas de estilo certamente a ferramenta *Checkstyle* os teria detectado.

Ao final as três ferramentas para Java ficaram empatadas com cada uma detectando três *bugs* diferentes.

Linguagem Python

Os resultados obtidos nas ferramentas Python foram bem abaixo do esperado, pois essas ferramentas só foram capazes de detectar o *bug* de variável não inicializada, enquanto

que as ferramentas Java e C foram capazes de encontrar muito mais erros. Esse resultado se deve principalmente ao fato dessas ferramentas serem mais focadas em problemas estilo ou segurança.

Os resultados do estudo para a linguagem Python podem ser encontrados na Tabela 11.

Tabela 11 - Resultados do estudo comparativo 1 para as ferramentas Python.

Bugs	**Pylint**	**Pyflakes**	**Bandit**	**pycodestyle**	**Python Taint**	**PyChecker**
Divisão por zero	✖	✖	✖	✖	✖	✖
Variável não inicializada	✔	✔	✖	✖	✖	✔
Acesso à posições fora do array	✖	✖	✖	✖	✖	✖
Acesso de uma posição a mais ou a menos em um array	✖	✖	✖	✖	✖	✖
Acesso à um arquivo que já foi fechado	✖	✖	✖	✖	✖	✖
Chamada de uma instrução inválida para o modo do arquivo	✖	✖	✖	✖	✖	✖
Chamada dupla de instrução para fechar arquivo	✖	✖	✖	✖	✖	✖
Arquivos não fechados após o uso	✖	✖	✖	✖	✖	✖
Falta de checagem de falha ao abrir arquivos	✖	✖	✖	✖	✖	✖
Instrução de *lock* que não é seguida de uma instrução de *unlock*	✖	✖	✖	✖	✖	✖

4.3.2 Considerações

A partir do estido foi possível constatar que as ferramentas C detectaram mais *bugs* do *Common Weakness Enumeration*, embora não possamos concluir que as ferramentas C serão melhores em todos os cenários. Em outros ocasiões as ferramentas Java e/ou Python

poderiam ter apresentado melhores desempenhos do que nesse estudo.

Comparando os resultados obtidos nessa monografia com os do artigo (CHATZIELEFTHERIOU; KATSAROS, 2011) é possível perceber algumas divergências nos resultados. Tais divergências portanto podem indicar duas coisas: que as ferramentas evoluíram desde a escrita do artigo ou que os *benchmarks* foram escritos de formas diferentes fazendo com que um determinado *bug* fosse encontrado ou não.

Por fim, nesse estudo foi possível perceber que as ferramentas se completam, principalmente para Java e C: padrões de bugs que são cobertos por uma ferramenta não são cobertos por outra, por exemplo. É perceptível que utilizar diversas ferramentas de análise estática simultaneamente em um projeto pode cobrir uma quantidade de *bugs* muito maior do que se estas fossem rodadas sozinhas.

4.4 Estudo Comparativo 2

No Estudo Comparativo 2 utilizamos ferramentas de análise estática para as linguagens Java, Python e C, submetidas a 28 projetos do *GitHub* e 2 da *Simbiose Ventures*. O objetivo principal desse estudo foi obter diversas métricas sobre as ferramentas existentes. As métricas utilizadas foram: interface com o usuário, nível de detalhamento dos padrões de bugs, quantidade de padrões de bugs, tempo de execução e consumo de memória.

Para esse estudo foram consideradas parte das ferramentas apresentadas na Seção 2.5 dessa monografia, as ferramentas *PyChecker*, *Python Taint*, *OCLint* e *Splint* foram descartadas pois elas não permitem rodar a análise em mais de um arquivo ao mesmo tempo e nesse estudo sempre rodamos as ferramentas sobre o projeto inteiro. O conjunto das ferramentas utilizadas nesse estudo pode ser encontrado na Tabela 12.

Tabela 12 - Ferramentas de análise estática utilizadas no estudo comparativo 2.

Java	**Python**	**C**
PMD	Pylint	Cppcheck
CheckStyle	Pyflakes	FlawFinder
FindBugs	Bandit	
SpotBugs	pycodestyle	

4.4.1 Programas analisados

Os projetos da *Simbiose Ventures* foram retirados do serviço *SlicingDice*[5]. O *SlicingDice* será uma boa base para o segundo estudo pois trará para o estudo comparativo um cenário de um produto real que não está no contexto *open source*. O projeto necessita ser altamente confiável, seguro e otimizado o que faz com que o uso de analisadores estáticos seja ainda mais importante. Do *SlicingDice* foram considerados dois projetos, o *S1Search* e a API (nessa monografia chamaremos a API simplesmente de *SlicingDice*). O *S1Search* é o *core* do *SlicingDice* sendo responsável por armazenar os dados e fazer as consultas a esses dados, a API é responsável não somente por receber as requisições *JSON* dos usuários e enviá-las ao *S1Search*, ela também é responsável por fazer controle das permissões do usuários. O *S1Search* será utilizado com as ferramentas Java, enquanto o *SlicingDice* com as ferramentas Python.

Para a escolha dos projetos no GitHub alguns critérios foram utilizados:

- Projeto tem que ser real;
- Projeto tem que ser popular (baseado na quantidade de estrelas do GitHub);
- O projeto deve ser complexo, ou seja, não deve ser uma simples calculadora ou página web;
- Projetos com maior quantidade de linhas de código devem ser priorizados.

Para todas as linguagens foram escolhidos programas complexos, populares (dentre os top 100 repositórios do GitHub) e que precisam de grande confiabilidade. Todos os programas foram extraídos do *GitHub* no dia 9 de outubro de 2017.

Nas tabelas 13, 14 e 15 são listados os programas que serão analisados nas linguagens Java, Python e C, respectivamente.

[5] O *SlicingDice* é um *data-warehouse* como serviço que permite a análise de grandes quantidades de dados sem que o usuário precise se preocupar com a infraestrutura por trás do serviço. O *SlicingDice* é concorrente direto de empresas como *Amazon RedShift* e *Google BigQuery*.

Tabela 13 - Programas analisados para a linguagem Java.

Programa	**Descrição**	**Arquivos**	**LOC**	**Repositório**
SlicingDice	Banco de Dados	497	69127	Código Confidencial
RoaringBitmap	Estrutura de Dados	203	48186	*GitHub*
Retrofit	Cliente HTTP	204	18298	*GitHub*
ZXing	Escaneador de código de barra	489	42903	*GitHub*
libGDX	Framework para desenvolvimento de jogos	2160	259002	*GitHub*
fastjson	*Parser* para JSON	2472	142999	*GitHub*
Storm	Sistema para computação distribuída	2259	255719	*GitHub*
PocketHub	Aplicativo Android	266	21066	*GitHub*
greenDAO	Mapeamento Objeto-Relacional para *SQLite*	244	19656	*GitHub*
Signal Android	Aplicativo Android	628	68322	*GitHub*

Tabela 14 - Programas analisados para a linguagem Python.

Programa	**Descrição**	**Arquivos**	**LOC**	**Repositório**
SlicingDice	Banco de Dados	104	48447	Código Confidencial
HTTPie	Cliente HTTP	47	4201	*GitHub*
Flask	Framework Web	71	8727	*GitHub*
Django	Framework Web	1896	218454	*GitHub*
Ansible	Automação para servidores	2373	345837	*GitHub*
Scrapy	Rastreador de páginas web	340	24795	*GitHub*
Keras	Biblioteca para *deep learning*	153	34544	*GitHub*
Certbot	Bot para habilitar regras https em servidores	251	32166	*GitHub*
Reddit	Rede Social	300	70596	*GitHub*
iPython	Shell para programar com Python	335	34515	*GitHub*

Tabela 15 - Programas analisados para a linguagem C.

Programa	**Descrição**	**Arquivos**	**LOC**	**Repositório**
CRoaring	Estrutura de dados	34	16381	*GitHub*
NGINX	Servidor HTTP	222	124416	*GitHub*
Redis	Banco de Dados	213	84569	*GitHub*
Arduino	Plataforma para desenvolvimento de hardware	95	29047	*GitHub*
OpenSSL	Plataforma para transporte de dados seguro	904	266912	*GitHub*
Curl	Cliente HTTP	417	104865	*GitHub*
ejoy2d	*Engine* para desenvolvimento de jogos	73	46735	*GitHub*
Torch	Navegador de Internet	50	17164	*GitHub*
OpenPilot	Plataforma para carros autônomos	13	7053	*GitHub*
ZStandard	Compressão de dados	106	56536	*GitHub*

4.4.2 Execução dos Estudos

Para cada um dos programas selecionados, os resultados correspondem à média de três execuções das ferramentas de análise estática, para análise o tempo de execução e o uso de memória foram computados. Para evitar que outros processos do sistema operacional influenciassem no tempo de execução foram utilizados o tempo de CPU (*user time*) e o tempo do sistema (*kernel time*) ao invés do tempo real. Para calcular essas métricas foi utilizado o programa *GNU Time*[6], para extrair o tempo de execução e o uso de memória das ferramentas que rodam através do terminal.

Como o *GNU Time* funciona essencialmente para ferramentas que rodam através do terminal, foi necessário uma outra técnica para medir o tempo de execução das ferramentas gráficas. Para essas ferramentas, todo o processo da análise foi gravado em vídeo, esse vídeo depois foi analisado para que o tempo de execução fosse o mais preciso possível. O *GNU Time* continuou sendo utilizado nas ferramentas com interface gráfica somente para extrair o uso de memória.

[6] `http://man7.org/linux/man-pages/man1/time.1.html`

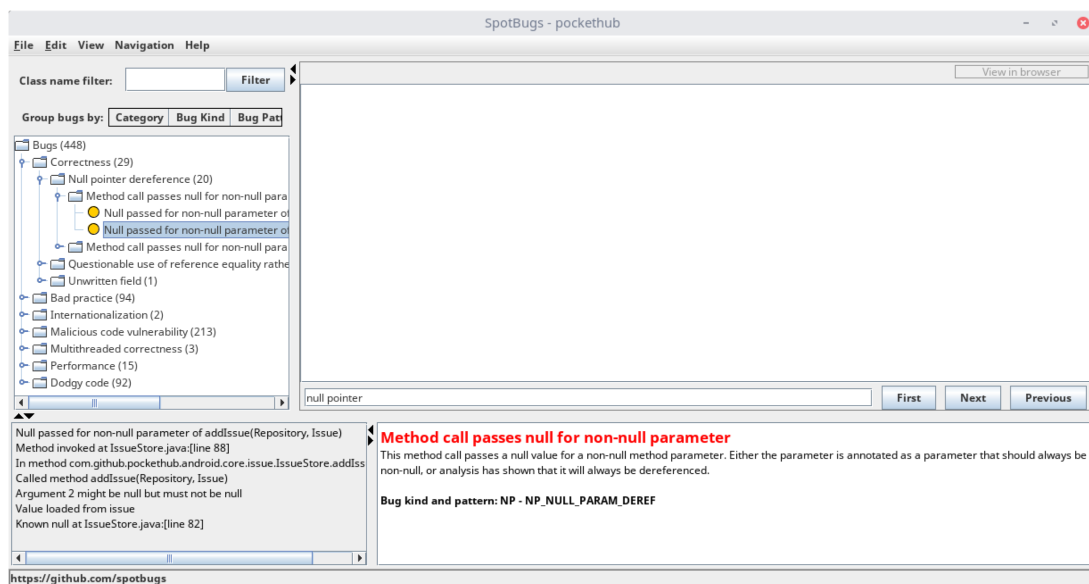
4.4.3 Resultado do Estudo

As ferramentas foram comparadas em relação aos seguintes aspectos: interface com o usuário, nível de detalhamento dos padrões de *bugs* apresentados, quantidade de padrões de *bugs* reportados, tempo de execução necessário para realizar uma análise completa e uso de memória durante a análise.

4.4.3.1 Interface com o usuário

Todas as ferramentas analisadas são bem parecidas em relação à interface com o usuário. Com exceção da *FindBugs* e *SpotBugs*, essas ferramentas são executadas via terminal e os resultados das análises são gravados em um arquivo. Vale salientar que, apesar da maioria dessas ferramentas serem executadas via terminal, elas possuem *plugins* para as mais variadas *IDEs*, fazendo com com que o programador não precise necessariamente usar o terminal. Na Figura 2 é apresentada a interface da ferramenta *SpotBugs* para Java.

Figura 2 - Interface gráfica da ferramenta *SpotBugs*.



Levando em consideração a interface com o usuário e os formatos de saída para as ferramentas nas linguagens Java, Python e C, elas foram classificadas nas tabelas 16, 17 e 18, respectivamente.

Tabela 16 - Comparação da interface com o usuário das ferramentas Java.

Ferramenta	**Interface Gráfica**	**Formatos de Saída**
PMD	Terminal	codeclimate, CSV, emacs, HTML, ideaj, summaryhtml, texto, texto com cor, textpad, vbhtml, XML, xslt, yahtml
Checkstyle	Terminal	XML e texto
FindBugs	Janela	XML e HTML
SpotBugs	Janela	XML e HTML

Tabela 17 - Comparação da interface com o usuário das ferramentas Python.

Ferramenta	**Interface Gráfica**	**Formatos de Saída**
Pylint	Terminal	texto, texto com cor, JSON e msvs
Pyflakes	Terminal	texto
Bandit	Terminal	CSV, HTML, JSON, texto e XML
pycodestyle	Terminal	texto

Tabela 18 - Comparação da interface com o usuário das ferramentas C.

Ferramenta	**Interface Gráfica**	**Formatos de Saída**
Cppcheck	Terminal	texto
FlawFinder	Terminal	texto e HTML

Diante dessa análise, podemos concluir que das ferramentas Java, a ferramenta *PMD* se destaca pois apresenta mais formatos de saída disponíveis. Por outro lado, no Python as ferramentas *Pylint* e *Bandit* possuem mais formatos de saída. Por fim, na linguagem C a ferramenta *FlawFinder* se destaca pois fornece um formato de saída em HTML.

4.4.3.2 Nível de detalhamento dos padrões de *bugs*

Outro ponto analisado nessa monografia foi o nível de detalhamento dos padrões de *bugs*. Neste quesito, as ferramentas foram comparadas considerando os seguintes aspectos:

- A ferramenta apresenta a explicação do padrões de *bug*?
- A ferramenta possui uma categorização dos padrões de *bugs*?
- A ferramenta exibe um *stacktrace*[7]?
- A ferramenta classifica os padrões de *bugs* de acordo com a severidade?
- A ferramenta apresenta o trecho do código onde o padrões de *bug* ocorreu?

Os resultados da comparação para as ferramentas Java, Python e C podem ser encontrados nas Tabelas 19, 20 e 21, respectivamente.

Tabela 19 - Nível de detalhamento dos padrões de *bugs* apresentados pelas ferramentas Java.

Ferramenta	**Explicação**	**Categoria**	**StackTrace**	**Severidade**	**Trecho do código**
PMD	Sim	Não	Não	Não	Não
Checkstyle	Simples	Sim	Não	Não	Não
FindBugs	Sim	Sim	Sim	Sim	Não
SpotBugs	Sim	Sim	Sim	Sim	Não

Tabela 20 - Nível de detalhamento dos padrões de *bugs* apresentados pelas ferramentas Python.

Ferramenta	**Explicação**	**Categoria**	**StackTrace**	**Severidade**	**Trecho do código**
Pylint	Simples	Não	Não	Não	Não
Pyflakes	Simples	Não	Não	Não	Não
Bandit	Simples	Não	Não	Sim	Sim
pycodestyle	Simples	Não	Não	Não	Não

[7]O *stacktrace* mostra quais linhas do código foram executadas até chegar em determinado ponto do código.

Tabela 21 - Nível de detalhamento dos padrões de *bugs* apresentados pelas ferramentas C.

Ferramenta	**Explicação**	**Categoria**	**StackTrace**	**Severidade**	**Trecho do código**
Cppcheck	Sim	Sim	Não	Não	Não
Flawfinder	Sim	Sim	Não	Não	Não

A partir dessa análise conclui-se que para Java as ferramentas *FindBugs* e *SpotBugs* são as que tem um maior nível de detalhamento. Para Python, a ferramenta mais bem detalhada é a *Bandit*, na Figura 3 é mostrado um exemplo de como os padrões de *bugs* são apresentados na ferramenta. Para C, as ferramentas *Cppcheck* e *Flawfinder* são bem parecidas quanto ao detalhamento dos padrões de *bugs* apresentados.

Figura 3 - Exemplo de resposta da ferramenta Bandit.

```
>> Issue: [B303:blacklist] Use of insecure MD2, MD4, or MD5 hash function.
   Severity: Medium   Confidence: High
   Location: certbot/account.py:59
58
59             self.id = hashlib.md5(
60                 self.key.key.public_key().public_bytes(
61                     encoding=serialization.Encoding.PEM,
62                     format=serialization.PublicFormat.SubjectPublicKeyInfo)
63             ).hexdigest()
```

Por fim, conclui-se que as ferramentas para Python e Java são mais completas que as ferramentas para C quanto ao nível de detalhamento dos padrões de *bugs* apresentados. As ferramentas Java se destacam pois, no geral, são mais detalhadas, enquanto das ferramentas Python somente a *Bandit* se destaca por ter um bom nível de detalhamento dentro das quatro analisadas.

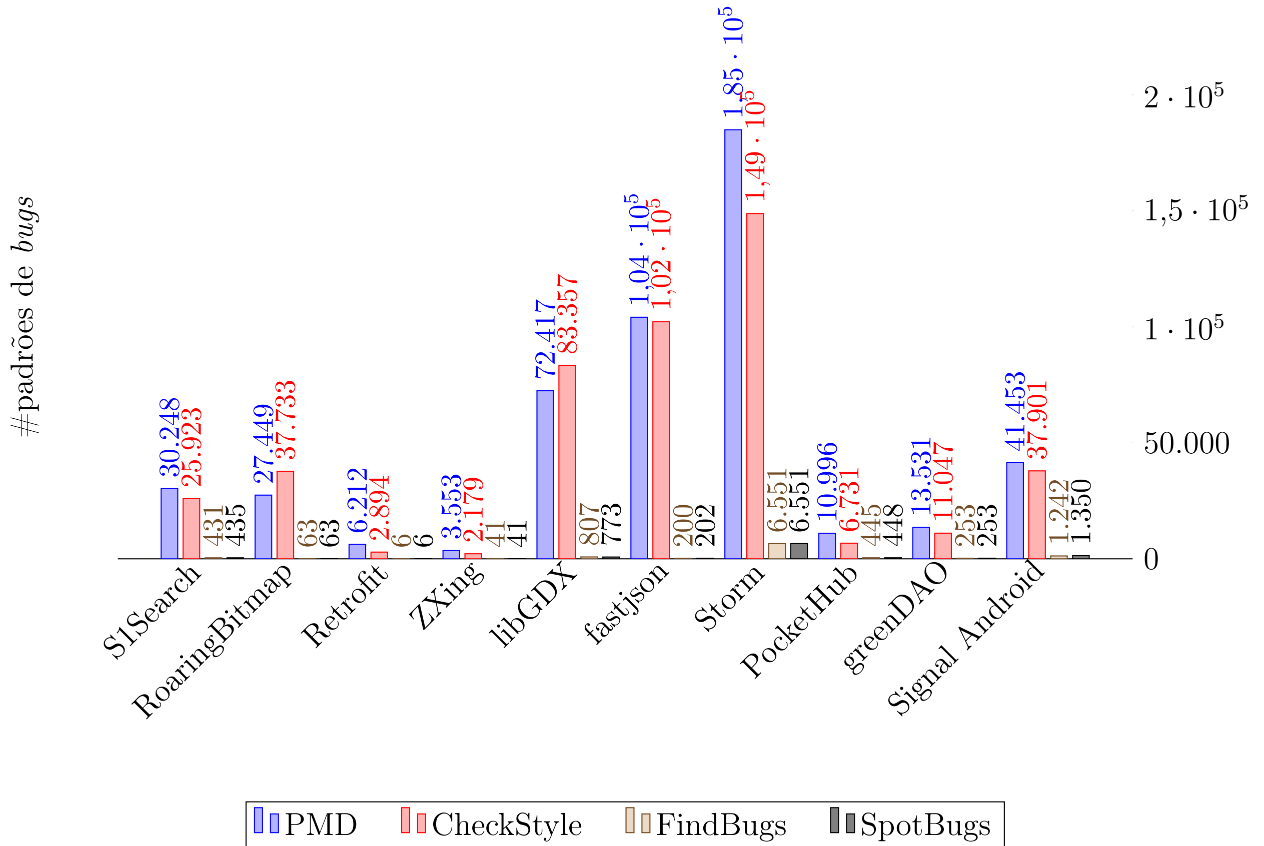
4.4.3.3 Quantidade de padrões de *bugs*

Nessa comparação foram computados quantos padrões de *bugs* foram reportados por cada uma das ferramentas. Vale salientar que a quantidade de padrões de *bugs* apresentados por uma ferramenta não indica que ela é melhor ou pior que outra, pois ferramentas diferentes podem apresentar objetivos diferentes. Como exemplo, algumas ferramentas podem se concentrar em problemas de segurança e bugs que são bem menos frequentes que problemas de estilo.

Linguagem Java

Ao analisar as ferramentas do Java (como pode ser visto na Figura 4) pode-se perceber que as ferramentas *PMD* e *CheckStyle* apresentam em média 50 vezes mais padrões de *bugs* que as ferramentas *FindBugs* e *SpotBugs*, isso se deve principalmente ao fato de que as duas primeiras ferramentas abrangem praticamente todos os tipos de erro, enquanto as duas últimas são focadas em erros de segurança e *bugs*.

Figura 4 - Quantidade de padrões de *bugs* para as ferramentas Java.



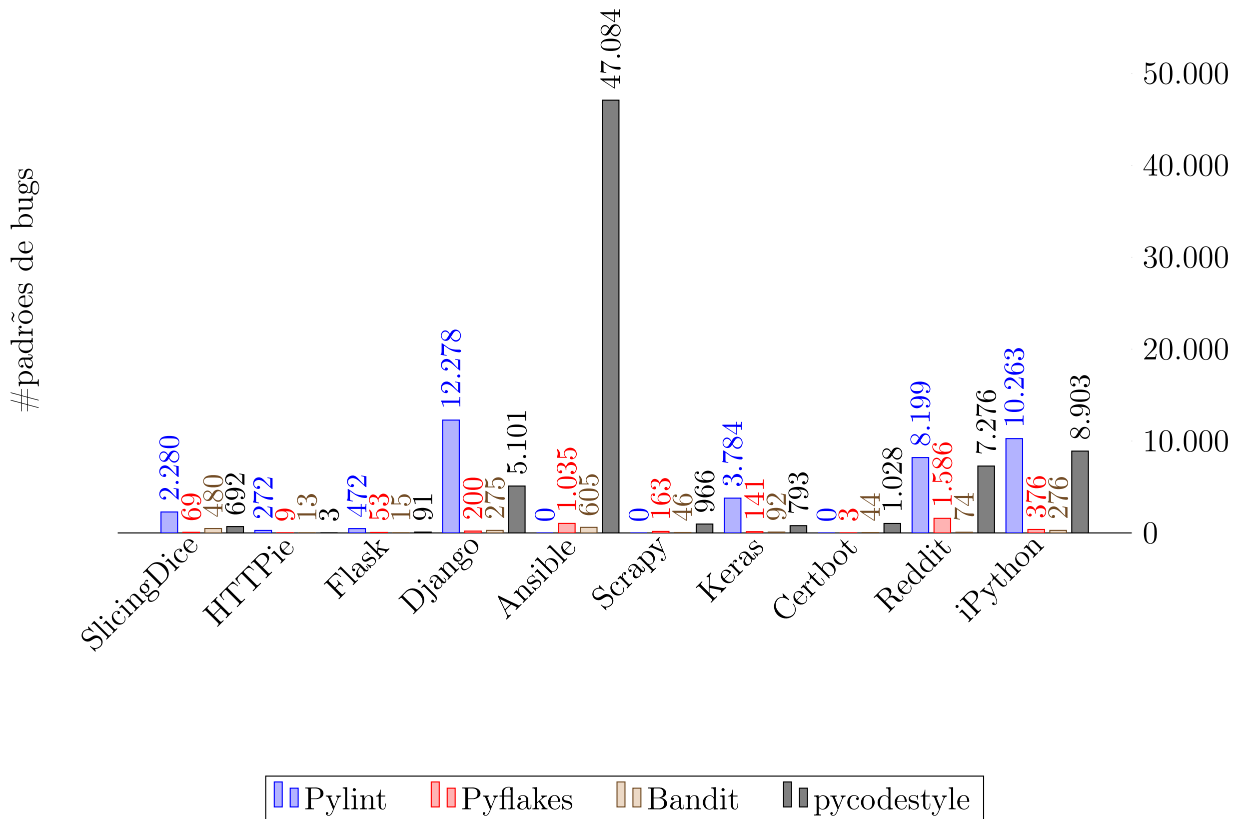
Ao analisar essas métricas, é possível perceber uma grande similaridade na quantidade de padrões de *bugs* apresentados pelas ferramentas *FindBugs* e *SpotBugs*, devido ao fato da *SpotBugs* ser baseada na *FindBugs* e usar um conjunto de regras parecido. Segundo o próprio site da *SpotBugs*, eles são os sucessores da ferramenta *FindBugs*.

Linguagem Python

Nos estudos realizados para as ferramentas para Python é importante ressaltar que não foi possível rodar as análises dos programas *Ansible*, *Scrapy* e *Certbot* na ferramenta *Pylint*, esta apresentou um erro de execução (*RecursionError: maximum recursion depth exceeded*). Para esses programas será colocado o valor 0 nos gráficos indicando que a ferramenta não conseguiu rodar as análises para o programa.

As ferramentas *Pyflakes* e *Bandit* geraram consideravelmente menos padrões de *bugs* do que as demais, principalmente devido ao fato das duas ferramentas terem um escopo bem fechado: a *Pyflakes* é focada somente em bugs, enquanto a *Bandit* é focada somente em problemas de segurança.

Figura 5 - Quantidade de padrões de *bugs* para as ferramentas Python.

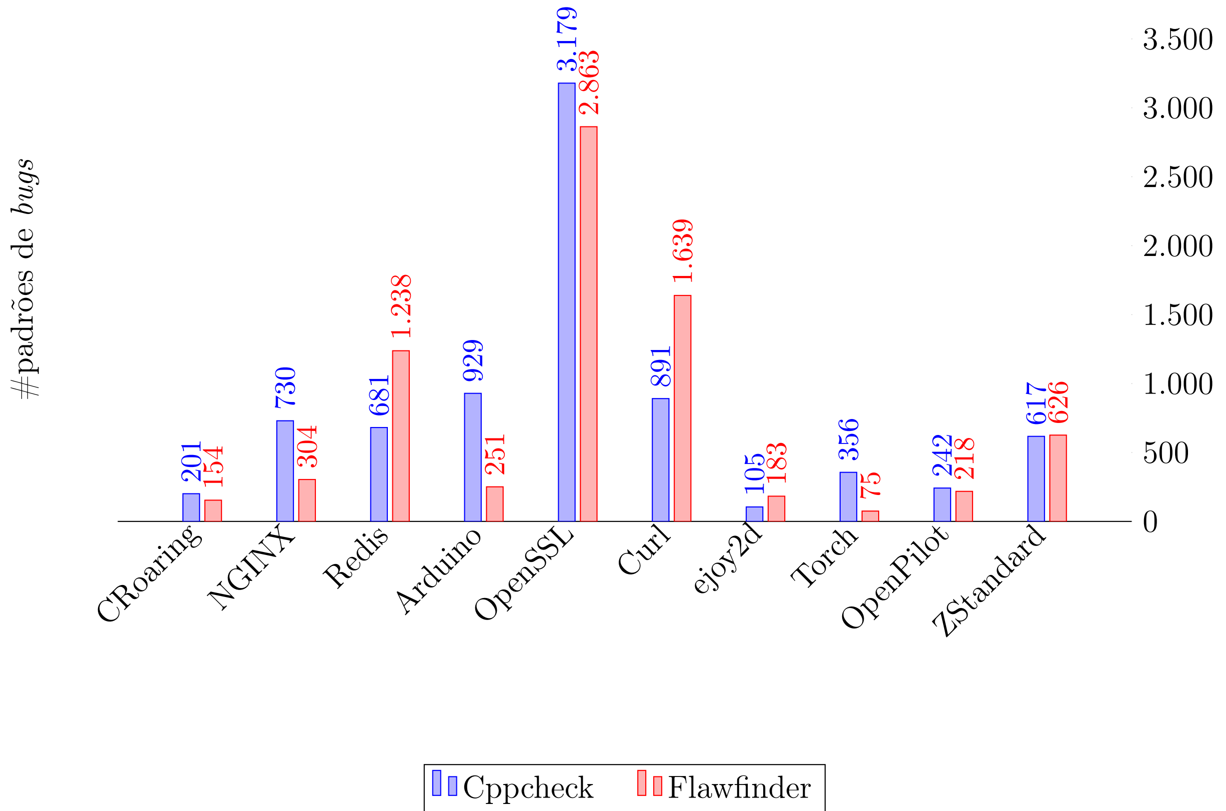


Por fim, é possível concluir que as ferramentas que analisam problemas de estilo no código apresentam muito mais padrões de *bugs* dos as que não os analisam, por esse motivo as ferramentas *Pylint* e *pycodestyle* apresentaram muito mais padrões de *bugs* do que as demais. Também é interessante notar, na Figura 5, a quantidade de padrões de *bugs* que o programa *Ansible* gerou na ferramenta *pycodestyle*, o que pode indicar que o projeto do *Ansible* não segue bem os padrões de código da linguagem Python.

Linguagem C

Nos estudos realizados para a linguagem C é possível perceber que a quantidade de padrões de *bugs* apresentados pelas duas ferramentas analisadas é bem similar (em alguns casos a diferença é somente de 11, como pode ser visto na Figura 6). Essa similaridade ocorre mesmo as ferramentas tendo focos diferentes: a *Cppcheck* concentra-se em achar *bugs* e códigos não usados, enquanto que a *Flawfinder* em problemas de segurança.

Figura 6 - Quantidade de padrões de *bugs* para as ferramentas C.



Por fim, é importante perceber que os projetos C não geraram tantos padrões de *bugs* quanto os projetos Java e Python, isso se deve principalmente ao fato das duas ferramentas C não serem focadas em estilo de código, deixando elas numa escala bem similar das ferramentas Java e Python que só são focadas em bugs e problemas de segurança.

4.4.3.4 Tempo de execução e consumo de memória

Durante o Estudo Comparativo 2, o tempo de execução e uso de memória foram calculados para todas as análises. As métricas para as ferramentas das linguagens Java, Python e C podem ser encontradas nas Tabelas 22 e 23; 24 e 25; e 26, respectivamente.

Os tempos foram calculados com base no tempo de usuário (tempo de uso da CPU) e no tempo de sistema (tempo no sistema operacional). O tempo total da análise corresponde ao tempo de usuário somado com o tempo do sistema. O consumo de memória é apresentado em megabytes e corresponde ao pico máximo de uso de memória durante a execução da ferramenta, enquanto que os tempos das análises são apresentados em segundos.

Para as ferramentas puramente gráficas somente será apresentado o tempo total de

execução, já que não é possível analisar o tempo da mesma forma que nas ferramentas que rodam no terminal.

Linguagem Java

As ferramentas Java foram as que mais consumiram memória RAM durante as análises, chegando a utilizar mais de 1 gigabyte de memória RAM em uma das análises. Quanto ao tempo das análises, as ferramentas Java ficaram numa escala bem parecida com as outras linguagens, ficando com um tempo máximo em torno de 260 segundos.

Tabela 22 - Tempo de execução e uso de memória nas ferramentas PMD e CheckStyle.

	PMD				**CheckStyle**			
	Tempo (em segundos)			Memória (em MB)	Tempo (em segundos)			Memória (em MB)
	Usuário	Sistema	Total		Usuário	Sistema	Total	
S1Search	80,05	0,92	80,97	887,45	23,45	0,43	23,88	408,43
RoaringBitmap	75,99	0,6	76,59	940,98	20,16	0,34	20,5	563,55
Retrofit	19,46	0,35	19,82	562,53	5,65	0,13	5,78	167,56
zxing	19,03	0,32	19,35	545,82	5,9	0,11	6,0	148
libGDX	98,98	0,91	99,89	933,42	29,34	0,49	29,83	560,60
fastjson	125,05	2,6	127,65	936,30	41,7	1,29	42,99	776,52
Storm	160,31	2,91	163,21	1108,88	53,16	1,26	54,42	884,02
PocketHub	39,86	0,67	40,52	820,72	11,58	0,33	11,91	222,58
greenDAO	38,8	0,65	39,45	830,74	11,09	0,28	11,37	230,74
Signal Android	80,87	0,98	81,85	882,74	25,38	0,58	25,95	414,09

Tabela 23 - Tempo de execução e uso de memória ferramentas FindBugs e SpotBugs.

	FindBugs		**SpotBugs**	
	Tempo (em segundos)	Memória (em MB)	Tempo (em segundos)	Memória (em MB)
S1Search	36	691,51	38,67	663,76
RoaringBitmap	37,33	711,70	36,67	710,49
Retrofit	8	366,25	9,33	427,30
zxing	13,50	453,83	13,33	468,11
libGDX	65,67	729,10	63,67	717,74
fastjson	34,33	678,93	31,33	672,36
Storm	258,33	1008,37	240	1081,95
PocketHub	39	526,73	39	532,67
greenDAO	25,67	487,88	25	532,78
Signal Android	135	719,45	88	728,66

No geral, entre as ferramentas Java, a ferramenta *CheckStyle* foi a que consumiu menos memória RAM e a ferramenta PMD foi a que consumiu mais, como pode ser visto na Figura 7.

Em questão de tempo total para realizar a análise, dentre as ferramentas que rodam pelo terminal a *CheckStyle* foi a que consumiu menos tempo (como pode ser visto na Figura 8). Por outro lado, as ferramentas que rodam via interface gráfica tiveram os tempos de execução bem parecidos, a *SpotBugs* foi ligeiramente melhor do que a *FindBugs* em alguns casos mas a diferença entre elas é quase imperceptível (como pode ser visto na Figura 9).

Figura 7 - Uso de memória nas ferramentas PMD, Checkstyle, FindBugs e SpotBugs.

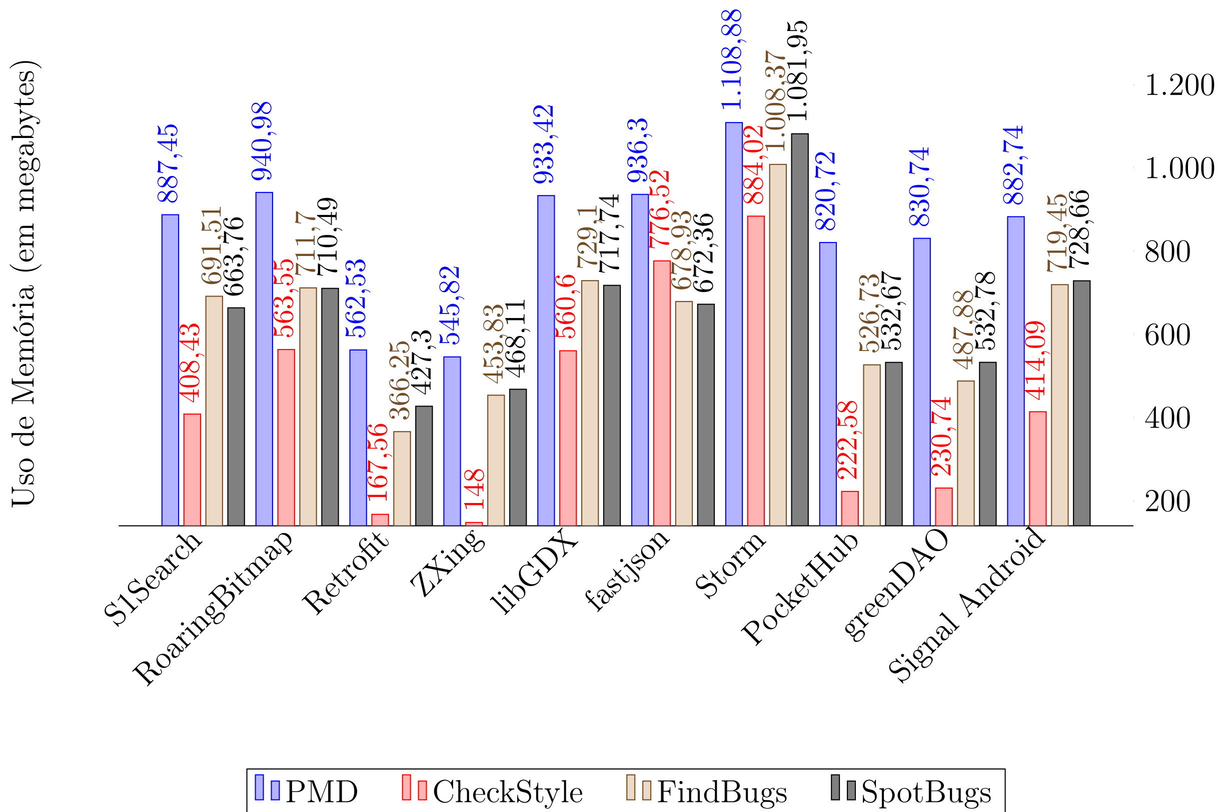


Figura 8 - Tempo de execução ferramentas PMD e CheckStyle (por projeto).

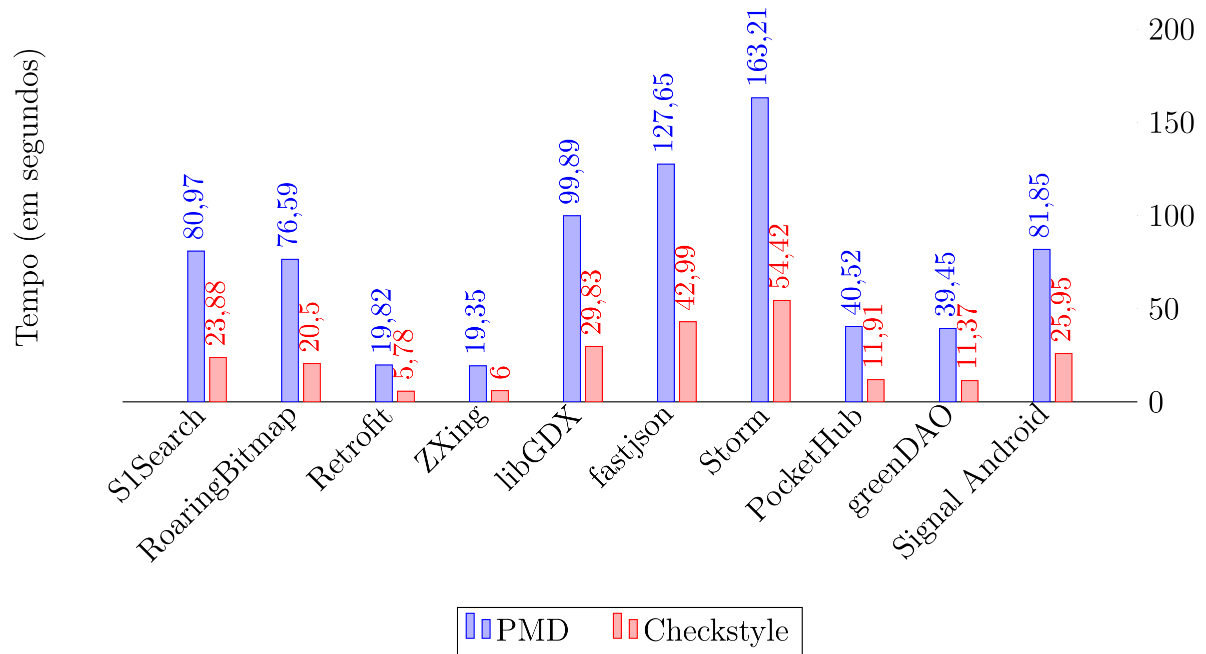
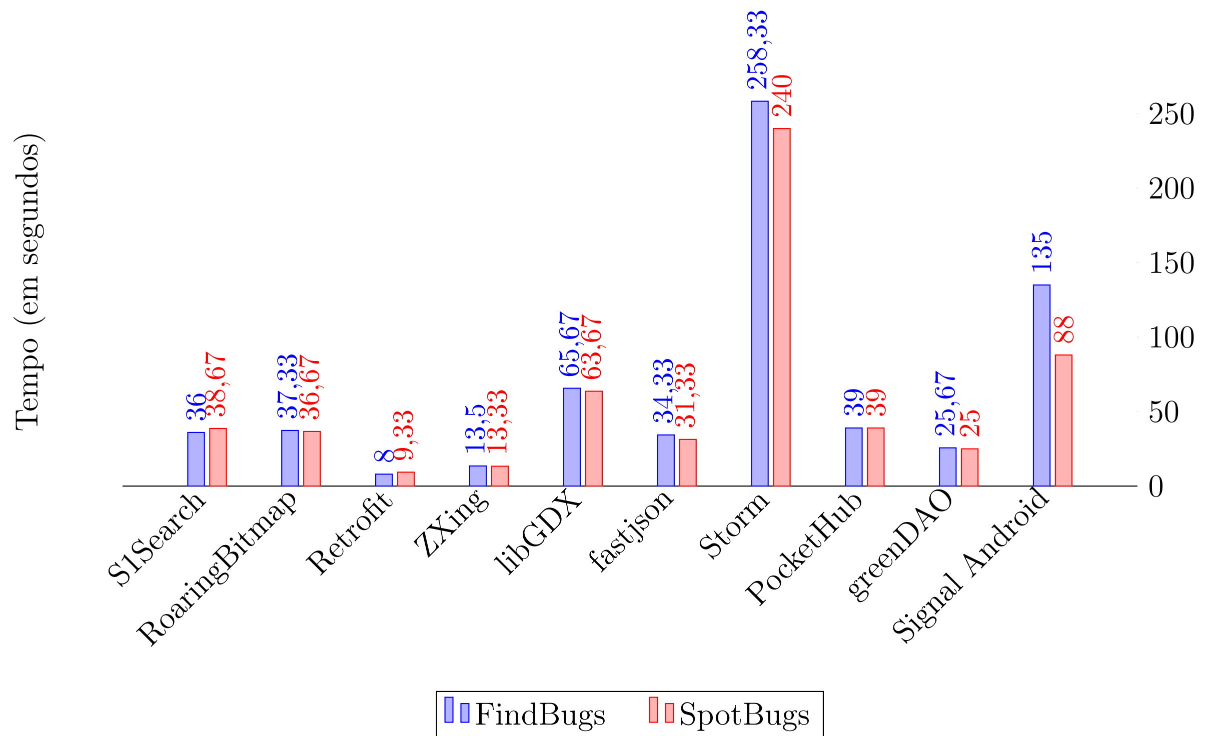


Figura 9 - Tempo de execução ferramentas FindBugs e SpotBugs (por projeto).



O tempo de execução também foi comparado levado em conta a quantidade de linhas de código (LOC) e a quantidade de padrões de *bugs* detectados. Como pode ser visto na

Figura 10 e 11 o tempo de execução desses estudos está diretamente relacionado com a quantidade de padrões de *bugs* reportados e não possui relação direta com a quantidade de linhas de código.

Figura 10 - Tempo de execução ferramentas Java (por quantidade de linhas).

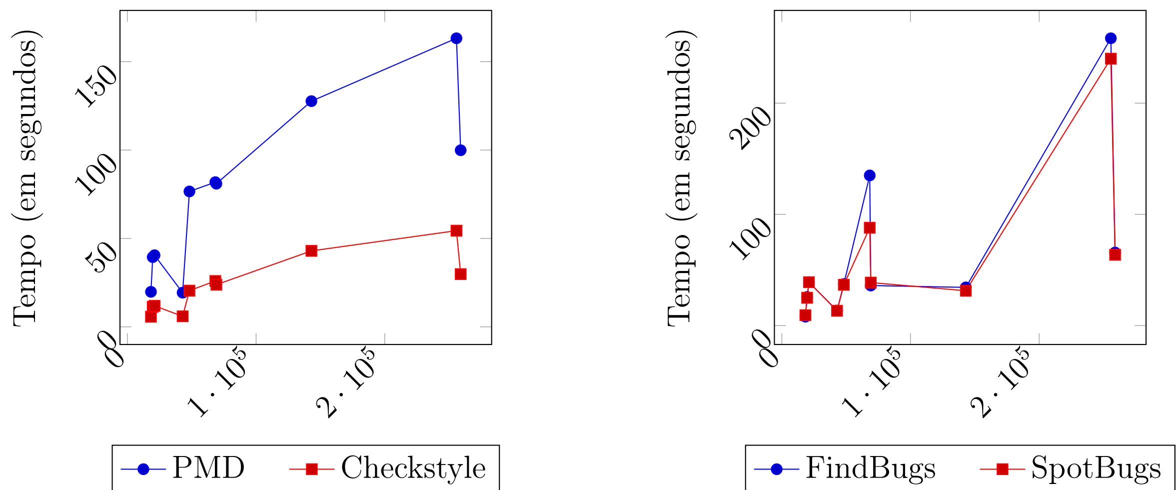
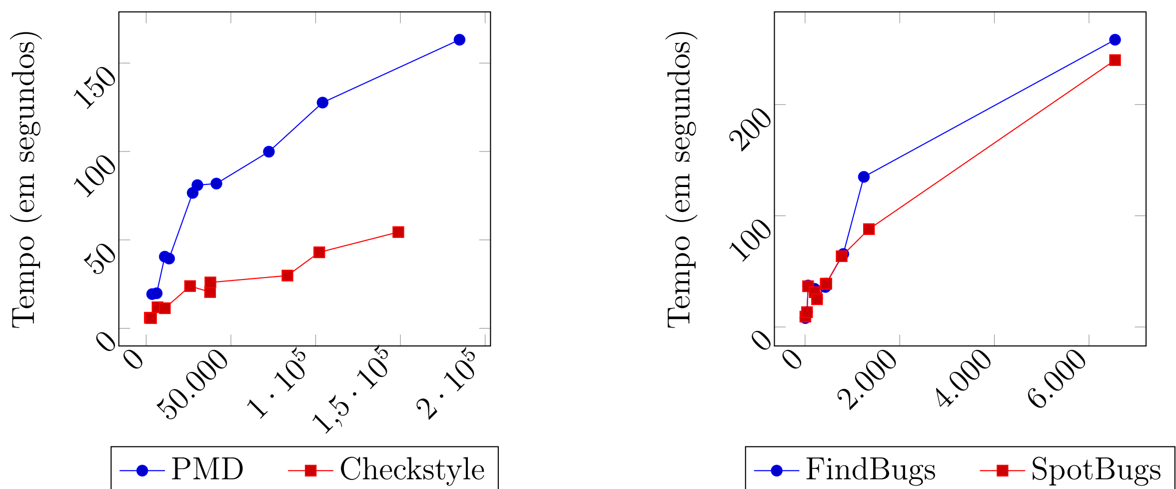


Figura 11 - Tempo de execução ferramentas Java (por quantidade de padrões de *bugs*).



Linguagem Python

As ferramentas para a linguagem de programação Python foram as que consumiram menos memória RAM. Com exceção da ferramenta Pylint, todas executaram as análises usando menos de 100 megabytes de memória. Por outro lado, o tempo de execução das ferramentas Python foi bem parecido ao das ferramentas para as outras linguagens.

Tabela 24 - Tempo de execução e uso de memória ferramentas Pylint e Pyflakes.

	Pylint				**Pyflakes**			
	Tempo (em segundos)			Memória	Tempo (em segundos)			Memória
	Usuário	Sistema	Total	(em MB)	Usuário	Sistema	Total	(em MB)
SlicingDice	36,37	1,58	37,95	136,20	1,84	0,06	1,9	100,68
HTTPie	8,18	0,36	8,54	102,38	0,12	0,01	0,13	22,04
Flask	8,5	0,4	8,9	101,32	0,19	0,01	0,2	25,76
Django	216,97	13,82	230,79	396,35	3,27	0,26	3,53	42,65
Ansible	-	-	-	-	12,53	0,15	12,68	48,82
Scrapy	-	-	-	-	0,6	0,01	0,61	22,69
Keras	47,38	1,13	48,51	174,38	01,06	0,02	01,09	41,72
Certbot	-	-	-	-	0,84	0,01	0,85	33,08
Reddit	172,53	16,62	189,14	340,16	3,27	0,09	3,36	78,78
iPython	136,65	3,15	139,8	638,34	1,35	0,03	1,38	32,99

Tabela 25 - Tempo de execução e uso de memória ferramentas Bandit e pycodestyle.

	Bandit				**pycodestyle**			
	Tempo (em segundos)			Memória	Tempo (em segundos)			Memória
	Usuário	Sistema	Total	(em MB)	Usuário	Sistema	Total	(em MB)
SlicingDice	7,83	0,05	7,87	48,48	7,14	0,0	7,14	13,23
HTTPie	0,94	0,02	0,96	28,07	0,48	0,0	0,49	12,04
Flask	1,33	0,02	1,35	29,12	0,8	0,0	0,8	12,13
Django	24,11	0,3	24,41	53,82	14,98	0,16	15,14	12,92
Ansible	98,26	0,24	98,51	59,10	58,45	0,1	58,55	12,96
Scrapy	3,99	0,04	04,03	38,39	2,32	0,01	2,33	12,16
Keras	7,45	0,02	7,47	48,69	04,04	0,0	04,04	12,42
Certbot	06,04	0,03	06,06	41,73	3,1	0,0	3,1	12,23
Reddit	13,8	0,07	13,88	48,07	12,49	0,01	12,5	13,01
iPython	23,18	0,08	23,26	48,79	6,72	0,02	6,74	14,89

Fazendo a comparação somente entre as ferramentas Python, a ferramenta *pycodestyle* foi a que utilizou menos memória RAM durante as análises, enquanto que a *Pylint* foi a que utilizou mais, como pode ser visto na Figura 12.

No que se diz respeito ao tempo total para realizar uma análise, a ferramenta *Pyflakes* foi a que consumiu menos tempo enquanto que a *Pylint* foi que consumiu mais tempo para rodar as análises, como pode ser visto na Figura 13.

Figura 12 - Uso de memória nas ferramentas Pylint, Pyflakes, Bandit e pycodestyle.

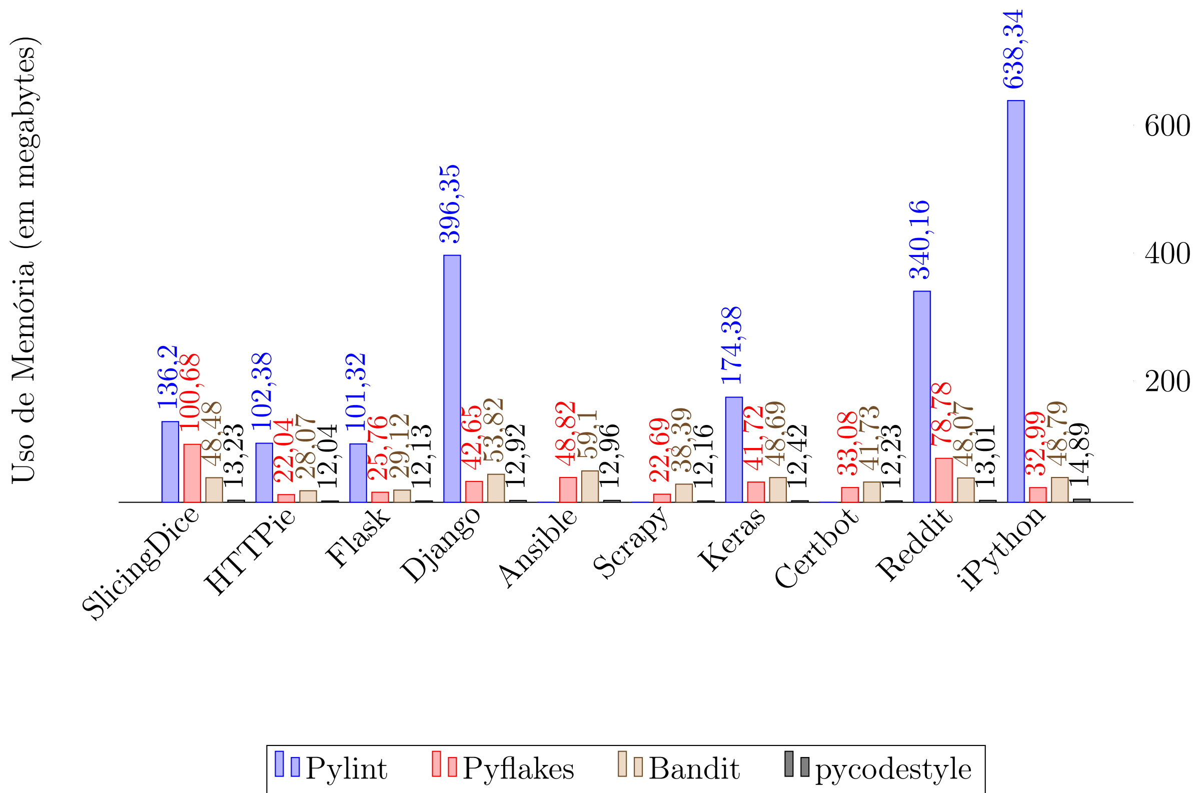
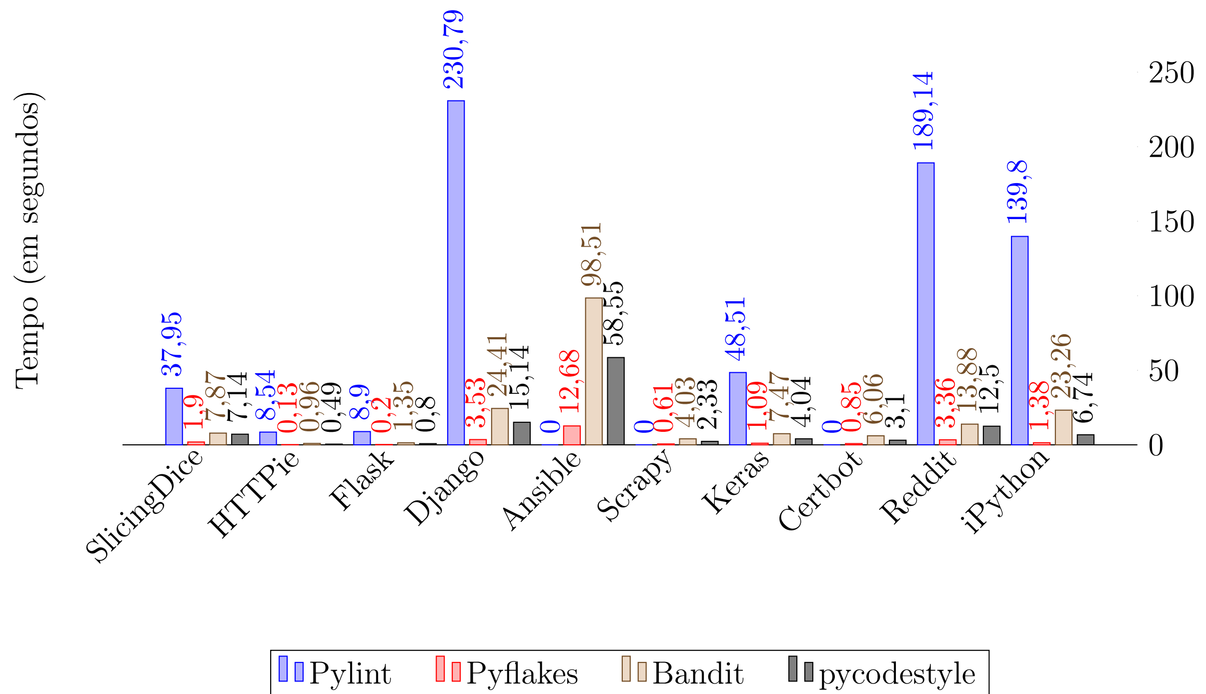


Figura 13 - Tempo de execução ferramentas Python (por projeto).



Diferentemente do que foi percebido nas ferramentas Java, as ferramentas Python

tiveram o tempo de execução diretamente atrelado à quantidade de linhas de código e à quantidade de padrões de *bugs* apresentados, como pode ser visto nas figuras 14 e 15.

Figura 14 - Tempo de execução ferramentas Python (por linhas de código).

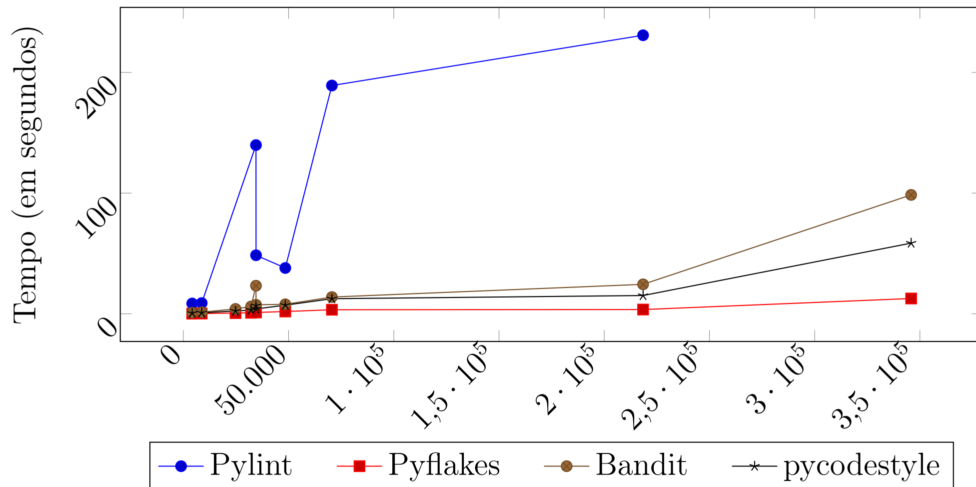
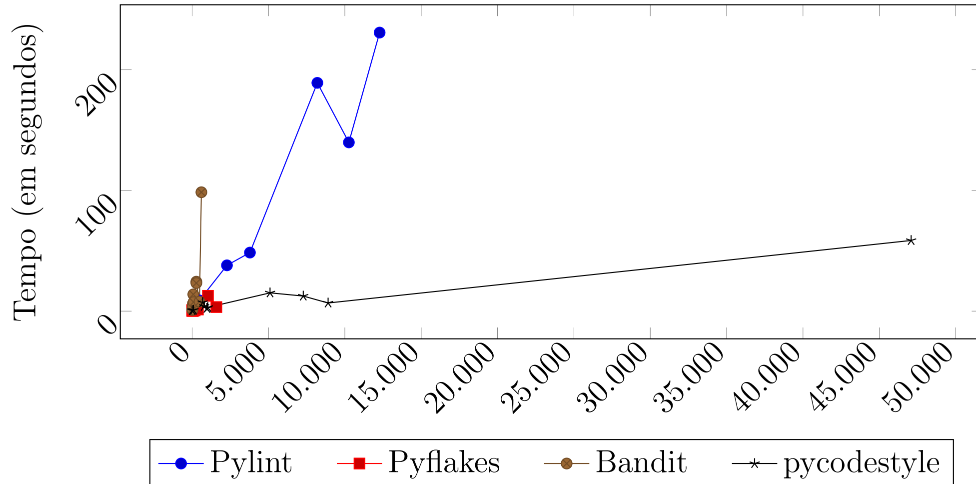


Figura 15 - Tempo de execução ferramentas Python (por quantidade de padrões de bugs).



Linguagem C

No quesito consumo de memória RAM, as ferramentas C ficaram classificadas entre as da linguagem Java e linguagem Python. A ferramenta *FlawFinder* no entanto merece destaque, como pode ser visto na Tabela 26, essa ferramenta não consumiu mais do que 20 megabytes de RAM em nenhuma das análises realizadas, comportamento bem parecido ao da ferramenta *pycodestyle*.

Tabela 26 - Tempo de execução e uso de memória ferramentas CppCheck e FlawFinder.

	CppCheck				**FlawFinder**			
	Tempo (em segundos)			Memória	Tempo (em segundos)			Memória
	Usuário	Sistema	Total	(em MB)	Usuário	Sistema	Total	(em MB)
CRoaring	6,96	0,02	6,98	21,33	0,62	0,03	0,65	13,66
NGINX	23,06	0,14	23,2	32,12	2,33	0,03	2,36	14,32
Redis	37,88	0,22	38,10	31,62	03,09	0,04	3,13	15,66
Arduino	9,1	0,32	9,42	193,70	3,9	0,08	3,98	16,81
OpenSSL	126,15	0,69	126,84	81,77	7,7	0,46	8,16	18,20
Curl	47,11	0,44	47,56	26,96	3,53	0,11	3,64	16,17
ejoy2d	289,23	0,12	289,35	129,24	1,57	0,01	1,58	16,05
Torch	127,46	0,39	127,85	699,06	0,49	0,01	0,5	13,76
OpenPilot	13,14	0,04	13,18	60,88	0,8	0,01	0,82	14,22
ZStandard	25,13	0,10	25,23	29,43	2,22	0,02	2,24	14,47

Entre as duas ferramentas C, a *FlawFinder* se destacou em ambos os quesitos, consumindo menos memória RAM e utilizando um menor tempo para rodar as análises, como pode ser visto nas Figuras 16 e 17, respectivamente.

Figura 16 - Uso de Memória nas ferramentas CppCheck e FlawFinder.

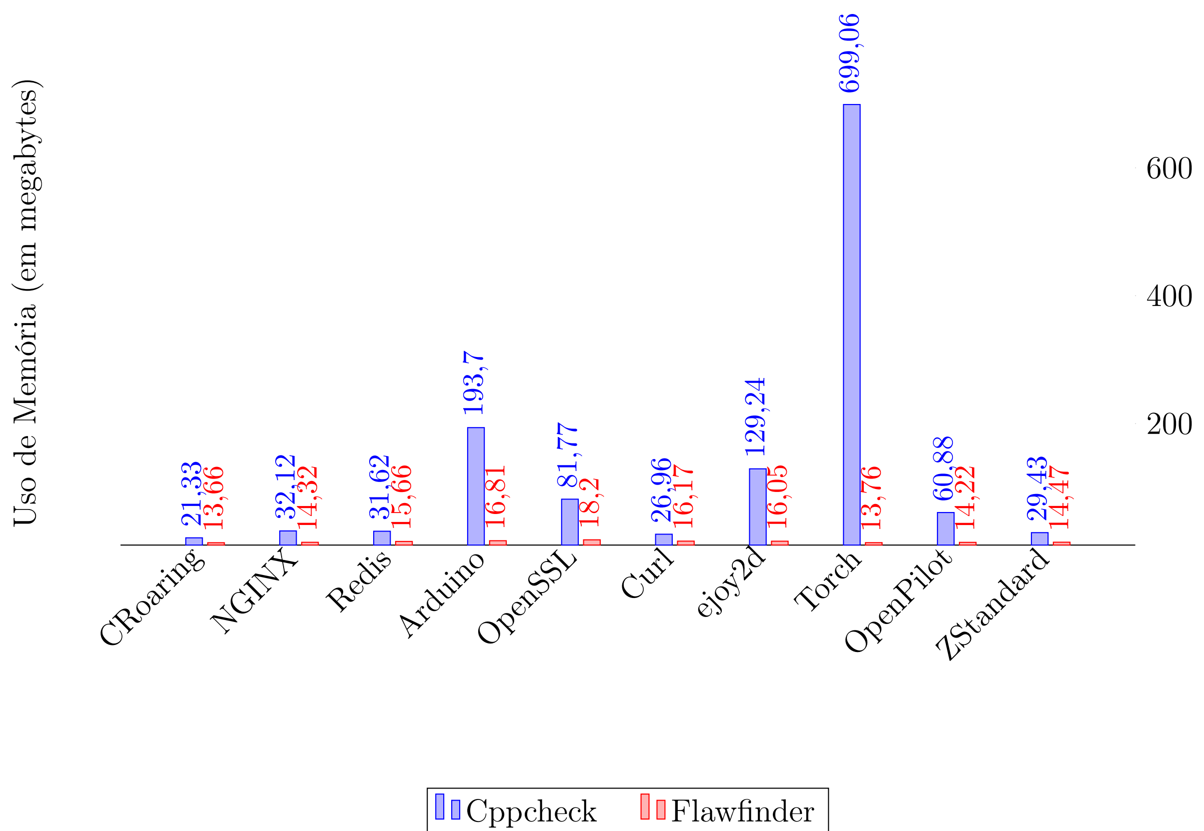
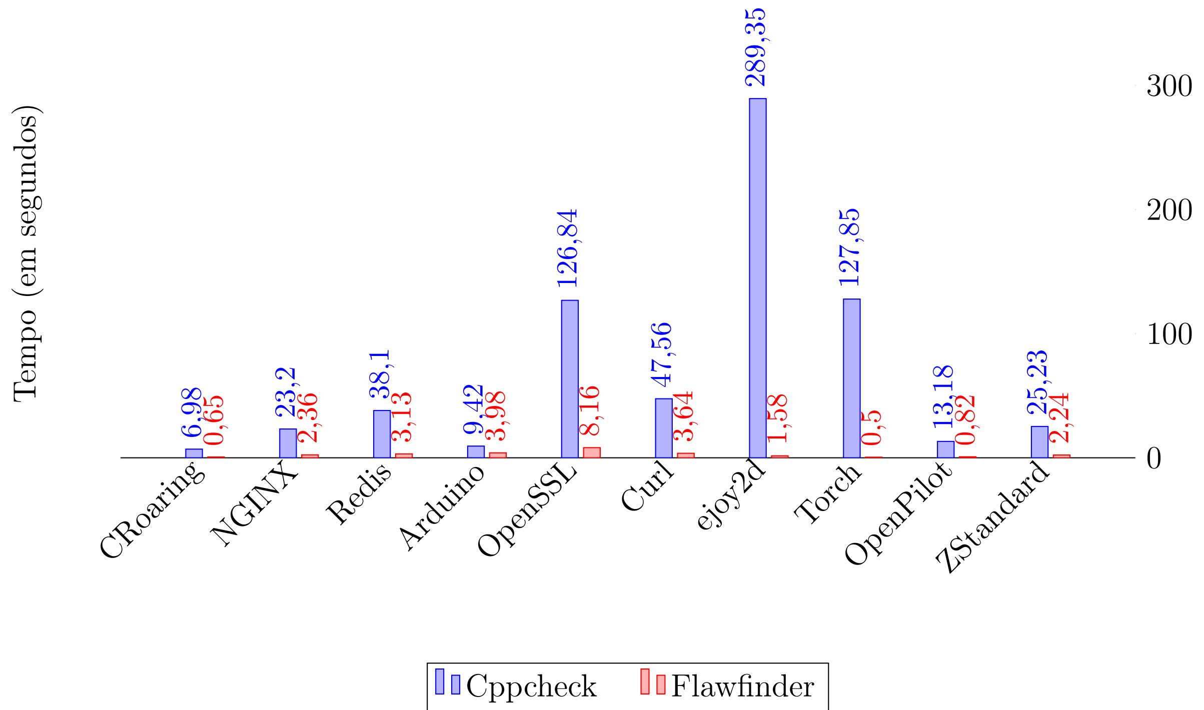


Figura 17 - Tempo de execução ferramentas C (por projeto).



Diferentemente das ferramentas Python e Java, as ferramentas para a linguagem C não apresentaram uma relação clara entre o tempo de execução, a quantidade de linhas ou a quantidade de padrões de *bugs*, como pode ser visto nas Figuras 18 e 19. Na ferramenta *FlawFinder*, o tempo de execução se manteve quase constante durante todos os testes. Esse comportamento não linear pode indicar que as ferramentas C possuem tempos de execução diferentes para padrões de *bugs* diferentes, ou seja, determinados padrões de *bugs* podem levar mais tempo para serem identificados.

Figura 18 - Tempo de execução ferramentas C (por linhas de código).

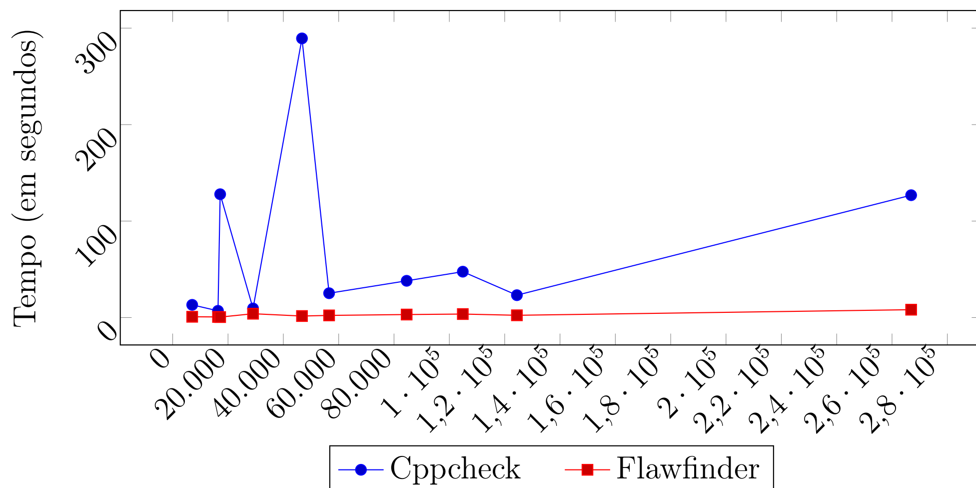
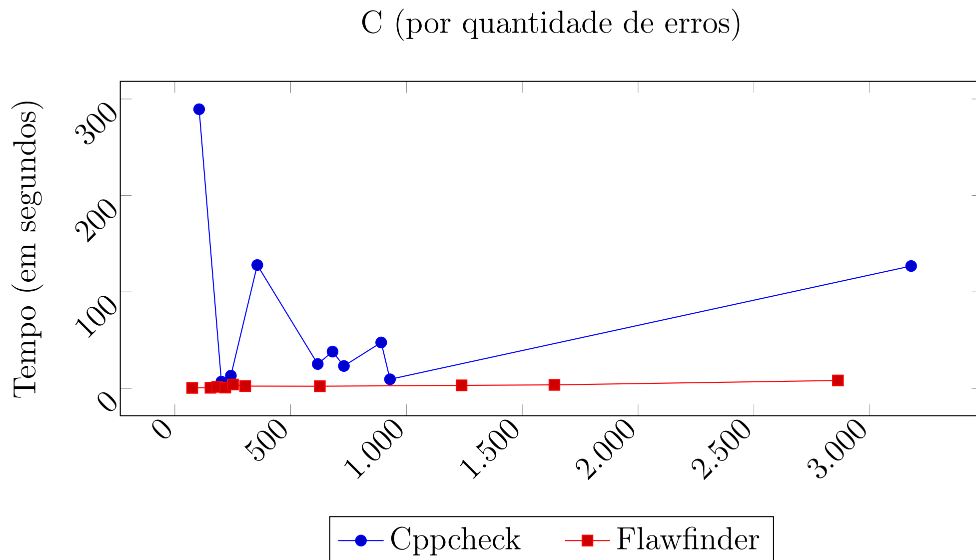


Figura 19 - Tempo de execução ferramentas Python (por quantidade de padrões de bugs).



4.4.4 Considerações

Ao fim do Estudo Comparativo 2 foi possível concluir que:

- Os tempos médios de execução entre as linguagens são bem parecidos;
- As ferramentas Python consomem menos memória RAM;
- As ferramentas Java tem um nível de detalhamento melhor dos padrões de *bugs*, permitindo ao programador interpretar melhor os resultados das análises;
- As ferramentas Java possuem uma interface de usuário mais completa que as das demais linguagens, oferecendo mais formatos de saída, por exemplo.

Vale salientar que apesar das ferramentas Python consumirem menos memória RAM, essa questão não deve ser o fator decisivo para escolher a linguagem Python, pois o consumo de memória das outras ferramentas continua sendo aceitável pelo fato de 1 gigabyte ser bastante pouco levando em conta que geralmente os computadores pessoais hoje possuem no mínimo 8 gigabytes de memória RAM.

Nesse estudo comparativo os falsos positivos e negativos não foram analisados, pois tal análise exigiria um conhecimento profundo de todos os projetos, o que seria inviável considerando o tempo disponível para a conclusão desta monografia. Desta forma, deixamos tal análise como um trabalho futuro.

O segundo estudo apresenta algumas limitações. Por exemplo, nas análises de tempo e de consumo de memória só foram realizadas três execuções para cada ferramenta e projeto. Além disso, diversos fatores que fogem ao nosso controle podem ter influenciado no tempo de execução dos programas. Dessa forma, os resultados apresentados aqui podem não ser totalmente precisos. Uma das formas de melhorar a precisão seria aumentar o número de repetições de cada análise.

4.5 Conclusão

Ao fim do estudo, foi possível perceber, dentro das ferramentas analisadas, que as linguagens Java e C são melhor providas de analisadores estáticos. Apesar das ferramentas Python no geral terem sido mais rápidas e consumido menos recursos do sistema operacional, as ferramentas para as linguagens Java e C foram capazes de detectar uma quantidade maior de padrões de *bugs* no catálogo *Common Weakness Enumeration*, isso ocorreu pois as ferramentas Java e C serem bem mais consolidadas que as ferramentas Python já que existem a muito mais tempo.

Por fim, conclui-se que as ferramentas Python analisadas são mais focadas em encontrar problemas de estilo, enquanto que as ferramentas para as demais linguagens são mais focadas em encontrar vulnerabilidades no software (com exceção da ferramenta *CheckStyle* para Java).

5 Considerações finais

Os softwares estão ficando cada vez mais complexos, dessa forma técnicas e ferramentas que ajudam o desenvolvedor a produzir código com qualidade estão ficando mais comuns. A análise estática está sendo cada vez mais adotada e consequentemente diversas ferramentas estão surgindo. Esse trabalho foi desenvolvido com o objetivo principal de apresentar as principais ferramentas existentes no mercado e compará-las.

A realização desse trabalho também foi motivada pelo constante uso de analisadores estáticos na empresa *Simbiose Ventures*, onde o autor dessa monografia atua como desenvolvedor de software. A forma de uso das ferramentas na empresa poderá ser melhorada com os conhecimentos adquiridos durante o desenvolvimento deste trabalho.

Na monografia foram apresentados diversos conceitos relacionados à análise estática que nos permitiram compreender melhor como as ferramentas funcionam. Esses conceitos foram de fundamental importância para o estudo comparativo, permitindo fazer uma escolha mais clara e inteligente dos aspectos que foram utilizados para classificar as ferramentas.

De fato, diante dos estudos realizados, as linguagens Java e C apresentaram os melhores resultados. As ferramentas para essas linguagens cobriram uma maior quantidade de erros do catálogo *Common Weakness Enumeration* (CWE) e, mesmo assim, realizaram as análises em tempos aceitáveis, permitindo que elas sejam utilizadas periodicamente durante o desenvolvimento de software sem interferir no desempenho do programador.

No entanto, a escolha de uma linguagem de programação ou de uma ferramenta de análise estática deve ser guiada pelos objetivos que se deseja atingir como o uso dela. Como as ferramentas possuem objetivos diferentes, não é possível indicar qual é a melhor para todos os casos, somente é possível chegar a esse tipo de conclusão diante dos objetivos da empresa/programador.

Pelos resultados obtidos nesse trabalho ficou claro que não se deve somente utilizar uma ferramenta: ao utilizar diversas ferramentas as chances de uma vulnerabilidade ser ignorada diminui bastante. Na empresa *Simbiose Ventures*, por exemplo, utilizamos praticamente todas as ferramentas *open source* para as linguagens Java e Python. No dia

a dia do desenvolvimento é perceptível como as ferramentas se completam e como utilizar somente uma delas não seria viável. Para facilitar a união de todas essas análises, utilizamos, na *Simbiose Ventures*, um serviço chamado *Codacy*[1], que permite rodar diversas ferramentas simultaneamente e periodicamente sobre o código e inclui as principais ferramentas disponíveis no mercado.

Por fim, esse trabalho contribuiu apresentando diversas comparações e métricas para ferramentas de análise estática nas linguagens Java, Python e C. Porém, a área relacionada ainda precisa ser bastante explorada. Ao ler os trabalhos relacionados e escrever essa monografia, ficou claro como um ou dois experimentos não são suficientes para classificar as ferramentas como um todo. Cada linguagem e cada ferramenta possui particularidades que exigem um número de experimentos significativos para que sejam devidamente analisadas.

Como trabalho futuro um estudo comparativo mais focado seria interessante, nesse estudo as análises deveriam ser feitas somente entre ferramentas que possuem o mesmo objetivo e padrões de *bugs* parecidos, dessa forma os resultados não seriam influenciados pela forma que a ferramenta funciona ou pela quantidade de padrões de *bugs* que ela detecta.

Um outro trabalho futuro poderia utilizar a quantidade de falsos positivos e falsos negativos nas comparações entre as ferramentas, tais métricas são um bom indicativo da eficácia de uma ferramenta de análise estática, porém calcular essas métricas exige um conhecimento profundo sobre o código que está sendo analisado pela ferramenta.

Ao final do desenvolvimento dessa monografia pudemos atingir todos os objetivos anteriormente definidos. Em especial, durante o estudo entendemos como as ferramentas de análise estática funcionam e como elas se relacionam a outros tipos de analisadores e a outras áreas da Computação. Dessa forma, fomos capazes de concluir o trabalho com uma avaliação da eficiência das ferramentas de análise estática em projetos e problemas reais.

[1] O serviço pode ser acessado através desse link: `https://www.codacy.com/`

Referências

CARDOSO, P. *SQL Injection: o que é e como evitar*. 2010. Disponível em: <http://www.zoomdigital.com.br/sql-injection-o-que-e-e-como-evitar/>. Acesso em: 25 set. 2017.

CHATZIELEFTHERIOU, G.; KATSAROS, P. Test-driving static analysis tools in search of c code vulnerabilities. In: IEEE. *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*. [S.l.], 2011. p. 96–103.

CHELF, B.; CHOU, A. The next generation of static analysis. 2007.

CODEEXCELLENCE. *What Is Static Analysis? And Why Is It Important To Software Testing*. 2012. Disponível em: <http://www.codeexcellence.com/2012/05/what-is-static-analysis-and-why-is-it-important-to-software-testing/>. Acesso em: 31 set. 2017.

CORNELL, D. Static analysis techniques for testing application security. *OWASP San Antonio*, 2008.

EMANUELSSON, P.; NILSSON, U. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, Elsevier, v. 217, p. 5–21, 2008.

FILHO, J. E. de A. et al. Um estudo sobre a correlaçao entre defeitos de campo e warnings reportados por uma ferramenta de análise estática. *IX Simpósio Brasileiro de Qualidade de Software*, p. 9–23, 2010.

GABRY, O. E. *Software Engineering - Software Process and Software Process Models (Part 2)*. 2017. Disponível em: <https://medium.com/omarelgabrys-blog/software-engineering-software-process-and-software-process-models-part-2-4a9d06213fdc>. Acesso em: 30 set. 2017.

GOMES, I. et al. An overview on the static code analysis approach in software development. *Faculdade de Engenharia da Universidade do Porto, Portugal*, 2009.

JOHNSON, S. *Lint, a C Program Checker*. Bell Laboratories, 1977. (Computing science technical report). Disponível em: <https://books.google.com.br/books?id=b8nWGwAACAAJ>.

KRATKIEWICZ, K.; LIPPMANN, R. Using a diagnostic corpus of c programs to evaluate buffer overflow detection by static analysis tools. In: *Proc. of Workshop on the Evaluation of Software Defect Detection Tools*. [S.l.: s.n.], 2005. p. 19.

MARTINS, E. *O que é cracker?* 2012. Disponível em: <https://www.tecmundo.com.br/o-que-e/744-o-que-e-cracker-.htm>. Acesso em: 30 set. 2017.

MORIMOTO, C. E. *Buffer Overflow*. 2005. Disponível em: <http://www.hardware.com-.br/termos/buffer-overflow>. Acesso em: 30 set. 2017.

NOVAK, J.; KRAJNC, A. et al. Taxonomy of static code analysis tools. In: IEEE. *MIPRO, 2010 Proceedings of the 33rd International Convention*. [S.l.], 2010. p. 418–422.

OWASP. *Static Code Analysis*. 2017. Disponível em: <https://www.owasp.org/index-.php/Static_Code_Analysis>. Acesso em: 31 set. 2017.

ROUSE, M. *dynamic analysis*. 2006. Disponível em: <http://searchsoftwarequality-.techtarget.com/definition/dynamic-analysis>. Acesso em: 27 out. 2017.

TEDESCO, K. *Complexidade ciclomática, análise estática e refatoração*. 2017. Disponível em: <https://www.treinaweb.com.br/blog/complexidade-ciclomatica-analise-estatica-e-refatoracao/>. Acesso em: 12 nov. 2017.

WHITTAKER, J. A. What is software testing? and why is it so hard? *IEEE software*, IEEE, v. 17, n. 1, p. 70–79, 2000.