Benchmarking entre Ferramentas de Análise Estática de Códigos Escritos em C

Rodrigo Leal, Guilherme Freitas e Francisco Airton Silva

Sistemas de Informação
Universidade Federal do Piauí (UFPI)

rocaleal98@gmail.com, guilhermedin@gmail.com, faps@cin.ufpe.br

Abstract. *Most systems in production have errors. To minimize this, Verification and Validation activities are indispensable. Static Code Analysis (SCA) is a software engineering technique for moderating errors, as well as for verifying styles, or both. Clang and Splint are examples of tools for the C language that aid in the reduction of errors using the AEC. This study aims at comparing these tools, evaluating and measuring the number of errors found and the execution time, by applying scenarios with different size codes with the same number of errors, two different machines executing the execution, and finally the 2 code analyzers. It was identified that the Algorithm used has more influence on the Execution Time of the ACS. As for the number of errors found, the Clang tool obtained maximum performance in the encounter of the errors, while it presented a greater cost of execution time when compared to the Splint.*

Resumo. *Grande parte dos sistemas em produção apresentam erros. Para minimizar isto, as atividades de Verificação e Validação são indispensáveis. A Análise Estática de Código (AEC) é uma técnica da Engenharia de Software para moderar erros, assim como para a verificação de estilos, ou ambos. Clang e Splint são exemplos de ferramentas para a linguagem C que auxiliam na redução de erros utilizando a AEC. Este estudo busca realizar a comparação dessas ferramentas, avaliando e mensurando o número de erros encontrados e o tempo de execução, mediante a aplicação dos cenários com códigos de tamanhos diferentes com mesmo número de erros, duas máquinas diferentes realizando a execução e por último, os 2 analisadores de código. Identificou-se que o Algoritmo usado possui maior influência sobre o Tempo de Execução da AEC. No que concerne ao número de erros encontrados, a ferramenta Clang obteve desempenho máximo no encontro dos erros, enquanto apresentou um custo maior de tempo de execução quando comparado com o Splint.*

1. Introdução

O desenvolvimento de um *software* pode se tornar difícil conforme as dimensões do problema. Além disso, o processo de desenvolvimento está sujeito a vários problemas, sejam eles internos ou externos, que resultam em um produto diferente do planejado. Diversos problemas podem ser identificados, porém, a maioria deles tem origem no erro humano [Maldonado and Delamaro 2016]. Com isso, cria-se a necessidade de maneiras para melhorar a qualidade do produto na indústria de *software*.

A Engenharia de *Software* trata da organização para produção de sistemas, e apresenta, então, processos para atingir a qualidade de produção que são: Verificação e Validação. A Verificação procura checar se o software atende a seus requisitos funcionais e não funcionais. Já, a Validação busca garantir que o *software* atenda às expectativas do cliente [Sommerville 2011].

Dentre as formas de se realizar a Verificação e Validação, destaca-se a Análise Estática de Códigos, uma técnica de Verificação que não envolve a execução de um programa e ajuda na captura e correção de erros humanos, desde imperfeições de sintaxe até imprecisões sérias que podem comprometer a segurança dos usuários; e os Testes Unitários usados para Validação de sistemas [Sommerville 2011].

Há várias ferramentas automáticas para realizar Análise Estática de Códigos na literatura para diversas linguagens de programação, como Ruby lint[1] para a linguagem Ruby, Pylint[2] para Python, FindBugs[3] para Java, e muitos outros analisadores.

A linguagem C se apresenta entre as linguagens mais populares quando se considera grande parte das áreas da computação. A Figura 1 apresenta, em ordem decrescente, as linguagens mais populares em uso na IEEE Xplore, Google e Github segundo [Spectrum 2015]. Considerando-se somente o critério *Embedded*, a linguagem C assume o topo do *ranking* de popularidade segundo a mesma pesquisa.

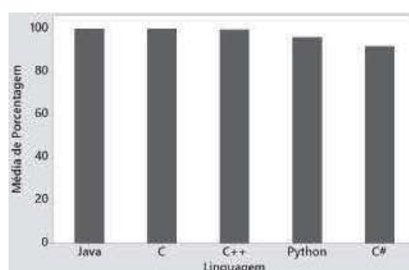

Figura 1. Popularidade de Linguagens de Programação.

O Clang e o Splint são analisadores da linguagem C frequentemente atualizados pela comunidade de desenvolvimento. O Clang é uma ferramenta, voltada para C e variantes, usada para encontrar erros no código e pode ser integrado com Interfaces de Desenvolvimento (IDEs) [Clang 2017]. O Splint, assim como o Clang, busca encontrar erros e vulnerabilidades de segurança. O Splint está na versão 3.1.2 e pode ser instalado em sistemas operacionais Linux e Windows [Splint 2017].

Este artigo estuda a influência dos algoritmos de análise no resultado final de um teste, e quais fatores tem maior impacto no tempo de execução dos analisadores Clang e Splint. Procura-se, assim, auxiliar engenheiros de *software* na seleção dos analisadores estáticos da linguagem C considerando o critério de desempenho em encontro de erros, e tempo gasto para execução.

O artigo é organizado com a seção 2 que apresentamos o estudo comparativo que retrata as ferramentas testadas, a metodologia empregada nos testes, a detecção

[1] https://github.com/YorickPeterse/ruby-lint
[2] https://www.pylint.org/
[3] http://findbugs.sourceforge.net/

de erros das ferramentas e o *Design of Experiments*, na seção 3 descrevemos os trabalhos relacionados com este estudo, e encerrando com a seção 4 com as conclusões.

2. Estudo Comparativo

Alguns erros que ocorrem em algoritmos não são visíveis para determinados compiladores. A Análise Estática de Código é a maneira para encontrar erros e reduzir defeitos em aplicações [Bardas et al. 2010]. A Análise Estática de Código pode ser feita com diversas técnicas, bibliotecas e algoritmos. Para a linguagem C, existem diversas ferramentas para realização dessa análise, a destacar Clang[4] e Splint[5] por conta de suas frequentes atualizações pela comunidade nos repositórios dessas ferramentas no GitHub [6].

O Clang é uma ferramenta para Análise Estática de linguagens como C, C++, Objective-C e variantes. Esse artefato de análise é usado para encontrar erros no código e pode ser integrado com Interfaces de Desenvolvimento (IDEs) [Clang 2017]. Clang tem código mantido no GitHub pela comunidade de desenvolvimento, e passa por constantes revisões e atualizações. O Clang possui algumas funcionalidades e metas disponíveis, entre elas, destaca-se a rápida compilação, baixo uso de memória, diagnósticos expressivos e código fonte de fácil entendimento.

Splint é uma ferramenta para verificação estatística de programas em C que busca encontrar erros e vulnerabilidades de segurança. O Splint está na versão 3.1.2 e pode ser instalado em sistemas operacionais Linux e Windows [Splint 2017]. A ferramenta Splint tem código-fonte mantido na plataforma GitHub, de forma aberta para a comunidade, e passa por atualizações frequentes [Splint 2017].

Para mensurar a performance do Splint e Clang, foram considerados os seguintes parâmetros: Número de erros encontrados, número de falsos positivos encontrados e tempo de execução das ferramentas.

Neste artigo, foram selecionadas, como estudos de caso, alguns erros frequentemente cometidos em projetos de *software* para avaliar a eficácia das ferramentas de análise estática, segundo [Terra 2008] há uma categorização de defeitos baseando-se no grau de impacto no projeto. Porém, esta categorização é baseada nos efeitos dos defeitos, e não nas causas ou nos tipos da ocorrência no código. As categorias são listadas na Tabela 1.

Tabela 1. Categorização de Erros de Código.

Índice	Categoria
1	Pane na aplicação
2	Falha lógica
3	Tratamento de erro insuficiente
4	Princípios da programação estruturada
5	Manutenibilidade do código

[4]https://github.com/llvm-mirror/clang
[5]https://github.com/ravenexp/splint
[6]https://github.com/

Considerando-se a Análise de Sensibilidade sobre os fatores que interferem no tempo de execução, utilizou-se três variáveis que podem influenciar diretamente no tempo gasto na execução. A Tabela 2 apresenta os fatores e níveis usados no experimento.

Tabela 2. Fatores e Níveis do Experimento.

Fatores	Níveis
Algoritmo	Splint Clang
Tamanho de Código	1200 linhas 300 linhas
Configuração da Máquina	Alta (Intel Core i5, 8GB de RAM, 1TB HD) Baixa (Intel Core 2 Duo, 4 GB RAM, 320 HD)

Em cada caso foram utilizados os mesmos algoritmos, com os mesmos números e tipos de erros executados 30 vezes, o que resultou em um número total de 240 experimentos ensaiados de forma aleatória. Para estudar o impacto desses fatores no tempo de execução, foi feita uma Análise de Sensibilidade usando DoE. Os resultados da Análise Sensibilidade pode ser observado no item B dessa mesma seção.

2.1. Detecção de Erros

O experimento para verificar a eficiência no encontro de erros foi organizado buscando apresentar as taxas percentuais de acerto para cada ferramenta de análise estática, considerando-se erros encontrados dentre os erros inseridos. Utilizou-se, no experimento, de 12 casos de erros aplicados para verificação e detecção do Splint e Clang. A Figura 2 apresenta o percentual de acerto para cada analisador estático.

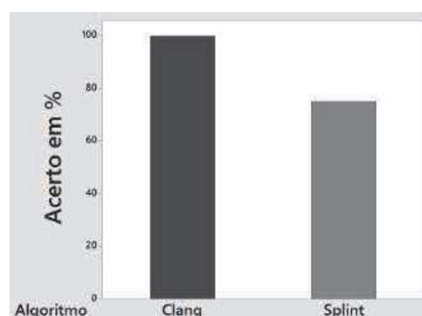

Figura 2. Quantidade de Erros Encontrados por cada Ferramenta.

O Splint detectou corretamente nove dos doze erros, obtendo uma taxa de acerto de 75%. O Splint foi capaz de detectar: repetições infinitas, bloco de condições declaradas erroneamente, uso de variáveis não declaradas e tipo de função inexistente. Entretanto, Splint não foi capaz de detectar os três seguintes tipos erros: falta de ponto-e-virgula, erro de alocação de memória pela declaração errada de ponteiros e utilização de função de biblioteca não declarada. Por outro lado o Clang conseguiu detectar todos os erros do teste.

Vale ressaltar que em ambos os casos de testes testados não houve ocorrência de falsos positivos, para melhor entendimento, consideramos falso positivo quando o algoritmo que rodamos os casos acusam erros em lugares onde não existem, assim, os erros encontrados realmente existiam no código. Um detalhamento dos erros utilizados nos experimentos com sua respectiva categorização estão dispostos na Tabela 3.

Tabela 3. Descrição de Categorias de Erros.

Índice	Descrição do experimento	Categoria	Splint	Clang
1	Utilização de variável não declarada	Princípio da programação estruturada	Sim	Sim
2	Não utilizar ponto-e-virgula ao fim da instrução	Princípio da programação estruturada	Não	Sim
3	Estrutura de repetição sem fim	Falha lógica	Sim	Sim
4	Bloco de condições declaradas sem parênteses	Princípio da programação estruturada	Sim	Sim
5	Tipo de função inexistente	Princípio da programação estruturada	Sim	Sim
6	Declaração errada da função scanf	Tratamento de erro insuficiente	Não	Sim
7	Utilização de biblioteca não declarada	Princípio da programação estruturada	Não	Sim
8	Declaração errada de struct	Falha lógica	Sim	Sim
9	Alocação dinâmica realizada de forma errada	Tratamento de erro insuficiente	Sim	Sim
10	Switch case com erro de break	Falha lógica	Sim	Não
11	Valores de retorno ignorados	Princípio da programação estruturada	Sim	Sim
12	Comentário errado realizado no código	Manutenibilidade do código	Sim	Sim

Na Tabela 3, observa-se a descrição de cada erro dos testes submetidos com suas respectivas classificações quanto ao grau de impacto sobre o projeto segundo [Terra 2008].

2.2. Análise de Sensibilidade com DoE

O Desenho de Experimento, do inglês *Design of Experiments* (DoE) permite conhecer o efeito de variações de parâmetros sobre o resultado, orientando, assim, a um processo de otimização. O DoE varia repetidamente cada um dos parâmetros, mantendo os demais constantes e observa as influências no resultado, conseguindo-se assim obter um *ranking* de influências [Minitab 2017]. Diversos gráficos podem ser usados para representação de estudos estatísticos, dentre eles, destaca-se o gráfico de Pareto.

O gráfico de Pareto apresentado na Figura 3 descreve a importância de um efeito por seu absoluto valor, desenhando uma linha vertical de referência no gráfico. Ao passo que o efeito prolonga essa linha, maior é a influência na variável dependente. A linha vermelha tracejada representa a magnitude mínima de efeitos estatisticamente significativos, considerando-se o critério de significância estatística α=0.05.

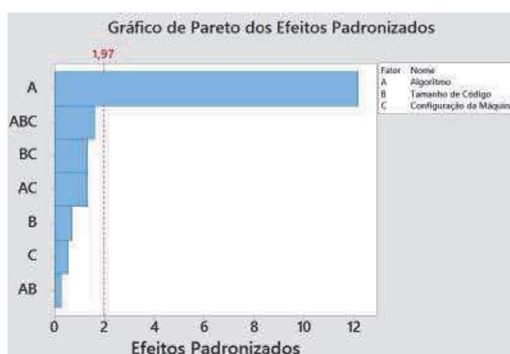

Figura 3. Influência dos Fatores sobre o Tempo de Execução.

Pode-se observar no gráfico que o fator com maior influência na variável dependente (Tempo de Execução) é o Algoritmo usado (representando pela letra A no gráfico). Variando-se a escolha entre o analisador Splint e Clang, obtêm-se a maior influência no tempo da análise estática de códigos.

Outras variáveis independentes como Tamanho de Código e Configuração da Máquina (representadas como B e C, respectivamente) não possuem influência estatística considerável na variável independente (Tempo de Execução).

Buscando complementar e melhor detalhar o estudo, utilizou-se o gráfico de Efeitos Principais (que pode ser visualizado na Figura 4). Este gráfico é usado para se demonstrar, detalhadamente, como um ou mais fatores categóricos influenciam em uma resposta contínua. A Figura 4 apresenta o gráfico de Efeitos considerando o experimento.

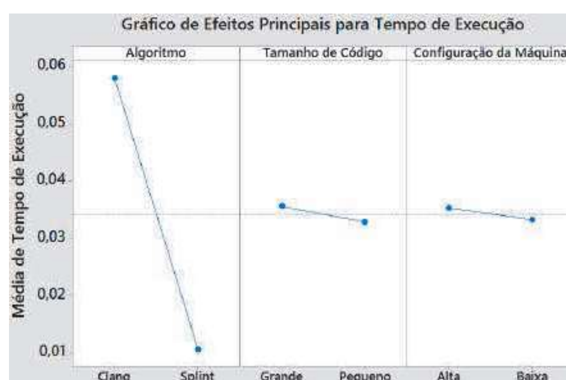

Figura 4. Gráfico de Efeitos Principais para Tempo de Execução.

O gráfico de Efeitos Principais apresenta inclinações (coloridas em azul no gráfico da Figura 4) entre as variáveis independentes estudadas. Quanto maior o esforço de inclinação, maior é a diferença da resposta contínua para as variáveis comparadas.

A ordem de esforço de inclinação é distinta para cada variável independente. Por ordem decrescente, o esforço de inclinação é organizado em Algoritmos, Configuração da Máquina e Tamanho de Código. Isso demonstra que o Tempo Médio de Execução é altamente distinto para os níveis de Algoritmos, regularmente distinto, para os níveis de Configuração de Máquina, e menor para o Tamanho de Código. A inclinação entre os níveis de Algoritmos é acentuada, o que retrata um Tempo Médio de Execução, elevadamente, distinto entre o analisador Splint e Clang. O analisador estático Splint teve um tempo inferior quando comparado com o Clang, com o tempo chegando a ser até 6 vezes menor e podendo gerar um impacto maior com códigos maiores.

3. Trabalhos Relacionados

Na Tabela 4 estão dispostos os trabalhos relacionados que seguem direção similar de pesquisa, ou dão base para este estudo.

Tabela 4. Comparativo entre Trabalhos Relacionados.

Trabalho	Linguagem	Análise de Sensibilidade
[Zitser et al. 2004]	C e C++	Não
[Manzoor et al. 2012]	C, C++ e Java	Não
[Chahar et al. 2012]	C, C++ e Java	Não
[Goseva-Popstojanova and Perhinschi 2015]	C, C++ e Java	Não
Nosso Trabalho	C	Sim

No geral foram encontrados artigos com estas características. Manzoor et al. (2012) procuraram medir a eficácia das ferramentas RELAY, RacerX e CheckThread em identificar *bugs*. Um ponto negativo deste trabalho foi a avaliação de ferramentas que ainda possuíam pouco uso da comunidade. Já Zitser et al. (2004) testaram cinco ferramentas que são: ARCHER, BOON, PolySpace C Verifier, Splint, e UNO. Os autores mensuraram as taxas de detecção e alarmes falsos (falsos positivos e negativos) das ferramentas em questão. Chahar et al. (2012) realizam análises de desempenho para comparar as ferramentas de verificação Flawfinder, Splint, e Cppcheck em um ambiente estático, bem como para comparar as ferramentas de verificação de códigos PMD, Findbugs, e Valgrind em um contexto dinâmico. Goseva-Popstojanova e Perhinschi (2015) realizaram um experimento de análise de códigos utilizando a Juliet benchmark, buscando um melhor entendimento de suas vantagens e de suas limitações.

Já neste trabalho, além de analisar o número de erros encontrados, diferenciando-se dos demais pelo estudo de DoE, verificando quais os fatores que tem maior impacto sobre o tempo de execução dos códigos testados.

- **Linguagem**: no quesito linguagem, foram encontrados Manzoor et al. (2012) que procurou medir o desempenho de ferramentas para as linguagens C, C++ e Java, assim como Chahar et al. (2012) e Goseva-Popstojanova e Perhinschi (2015) , já o Zitser et al. (2004) só contemplou as linguagens C e C++.
- **Análise de Sensibilidade**: com exceção deste trabalho, nenhum dos outros citados acima realiza análise de sensibilidade.

4. Conclusões

Com relação ao número de erros encontrados na execução dos testes e ao número de falsos positivos, podemos concluir que a ferramenta Clang superou a ferramenta Splint nestes quesitos, pois foi capaz de encontrar todos os erros propostos nos testes, obtendo completo êxito, enquanto o Splint não conseguiu encontrar todos, obtendo uma taxa de acerto de 75%. Com estes resultados, pode-se concluir que a ferramenta Clang teve um melhor resultado no quesito eficácia. Em outro experimento apresentado, percebe-se que o tempo de execução sofreu influência dos três fatores estudados, o que pode ser comprovado com a Análise de Sensibilidade apresentada na seção 2.2. O fator que teve maior influência sobre o tempo de execução foi o Algoritmo, acima de configuração da máquina e tamanho de código, o que pode ponderar com maior valor a escolha adequada do algoritmo para aumento da eficiência da análise estática. Atrás do fator de algoritmo, a configuração da máquina exerceu pouca influência sobre os resultados, se comparado com o fator algoritmo, sendo responsável por pouco alterar o tempo de processamento durante variação das propriedades da máquina de

análise. Mas embora sua relevância ser pouca ainda se sobre-sai quando comparado ao tamanho do código.

Referências

Bardas, A. G. et al. (2010). Static code analysis. *Journal of Information Systems and Operations Management*, 4(2):99–107.

Chahar, C., Chauhan, V. S., and Das, M. L. (2012). Code analysis for software and system security using open source tools. *Information Security Journal: A Global Perspective*, 21(6):346–352.

Clang (2017). Clang: a c language family frontend for llvm. In *Clang: a C language family frontend for LLVM*.

Goseva-Popstojanova, K. and Perhinschi, A. (2015). On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33.

Maldonado, C, J. and Delamaro, E. M. (2016). *Introdução ao Teste de Software*. Elsevier, 2th edition.

Manzoor, N., Munir, H., and Moayyed, M. (2012). Comparison of static analysis tools for finding concurrency bugs. In *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, pages 129–133. IEEE.

Sommerville, I. (2011). *Engenharia de Software*. Pearson, 9th edition.

Spectrum, I. (2015). Interactive: The top programming languages 2015. In *Interactive: The Top Programming Languages 2015*.

Splint (2017). Splint - secure programming lint. In *Splint - Secure Programming Lint*.

Terra, R. (2008). Splint - secure programming lint. In *Ferramentas para Análise Estática de Códigos Java*.

Zitser, M., Lippmann, R., and Leek, T. (2004). Testing static analysis tools using exploitable buffer overflows from open source code. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 97–106. ACM.