UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Avaliação do uso de perfis de desempenho aplicados no teste de software em diferentes domínios de aplicação

Thiago de Jesus Oliveira Durães

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP
SÃO CARLOS

Thiago de Jesus Oliveira Durães

Avaliação do uso de perfis de desempenho aplicados no teste de software em diferentes domínios de aplicação

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *EXEMPLAR DE DEFESA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Paulo Sergio Lopes de Souza

USP – São Carlos
Julho de 2022

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

D947a

Durães, Thiago de Jesus Oliveira
Avaliação do uso de perfis de desempenho aplicados no teste de software em diferentes domínios de aplicação / Thiago de Jesus Oliveira Durães; orientador Paulo Sergio Lopes de Souza. -- São Carlos, 2022.
119 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2022.

1. Teste de software. 2. Monitoração. 3. Agrupamento. 4. Perfis de desempenho. 5. Aprendizado de máquina não supervisionado. I. Souza, Paulo Sergio Lopes de , orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Thiago de Jesus Oliveira Durães

Evaluating the use of performance profiles applied in software testing in different application domains

Master dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP, in partial fulfillment of the requirements for the degree of the Master Program in Computer Science and Computational Mathematics. *EXAMINATION BOARD PRESENTATION COPY*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Paulo Sergio Lopes de Souza

USP – São Carlos
July 2022

AGRADECIMENTOS

Agradeço ao meu orientador, Prof. Dr. Paulo Sergio Lopes de Souza pelas experiências e ensinamentos durante o período do mestrado.

Agradeço aos colegas Vitor, Diego, João, Endi e demais colegas do ICMC, e à Profa. Dra. Simone Do Rocio Senger de Souza, pelo suporte e sugestões durante o desenvolvimento do trabalho.

Agradeço o fundamental apoio financeiro da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001

“The Universe is under no
obligation to make sense to you.”
(Neil deGrasse Tyson)

RESUMO

DURÃES, T. J. O. **Avaliação do uso de perfis de desempenho aplicados no teste de software em diferentes domínios de aplicação**. 2022. 119 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2022.

O processo de desenvolvimento de software é uma tarefa complexa, apoiada pela engenharia de software. Apesar da evolução da engenharia de software, os sistemas ainda falham por defeitos inseridos no desenvolvimento. O teste de software tenta revelar defeitos, usando técnicas de teste bem conhecidas, as quais são usualmente baseadas em requisitos funcionais dos sistemas. Requisitos não funcionais, como o uso de recursos, também podem ser usados no teste, pois apontam comportamentos não esperados de uso de processador, memória, e entrada e saída. A metodologia de teste *Tricorder*, em desenvolvimento no ICMC/USP, agrupa perfis de desempenho (uso de recursos), de modo a automatizar a detecção de padrões anômalos de comportamento de sistemas de software. A *Tricorder*, que atua de maneira complementar aos testes já realizados no sistema, usa algoritmos de aprendizado de máquina não supervisionados, e já foi inicialmente avaliada, com sucesso, na detecção de defeitos funcionais de aplicações que processam *streams* de dados. O principal objetivo deste projeto é estender a avaliação da metodologia de teste *Tricorder* em outros domínios de aplicação que não apenas o processamento de *streams* de dados. Espera-se, ao atingir este objetivo principal, identificar eventuais ajustes na metodologia *Tricorder* que possam melhorar a sua automatização, eficácia e eficiência. A metodologia usada neste projeto considerou quatro diferentes experimentos para revelar defeitos em sistemas de criptografia, métodos de ordenação, processamento de vídeo e simulações de fluídos. Os defeitos analisados foram baseados em taxonomias de defeitos existentes na literatura e, sempre que possíveis, foram usados defeitos reais, relatados por seus desenvolvedores. As métricas usadas nas avaliações considerou a capacidade de revelar defeitos e a quantidade de falsos positivos e falsos negativos. Os resultados dos experimentos mostram que a *Tricorder* consegue revelar com sucesso defeitos de software nos domínios de aplicação analisados, para a grande maioria das execuções com defeitos. Este trabalho contribui com a atividade de teste de software, em particular com a redução do custo desta atividade, pois estender o uso da *Tricorder*, uma metodologia que automaticamente detecta defeitos em programas, sem a necessidade criar novos casos de teste e de oráculos.

Palavras-chave: Teste de software, Monitoração, Agrupamento, Perfis de desempenho, Aprendizado de máquina não supervisionado.

ABSTRACT

DURÃES, T. J. O. **Evaluating the use of performance profiles applied in software testing in different application domains**. 2022. 119 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2022.

The software development process is a complex task, supported by software engineering. Despite the evolution of software engineering, systems still fail due to defects inserted in the development. Software testing attempts to reveal defects using well-known testing techniques, which are usually based on functional requirements of the systems. Non-functional requirements, such as resource usage, can also be used in testing, as they point out unexpected processor, memory, and input/output usage behaviors. The *Tricorder* test methodology, under development at ICMC/USP, groups performance profiles (resource usage) in order to automate the detection of anomalous behavior patterns in software systems. *Tricorder*, which works in a complementary way to the tests already carried out on the system, uses unsupervised machine learning algorithms, and has already been successfully evaluated in the detection of functional defects in applications that process data streams. The main objective of this project is to extend the evaluation of the *Tricorder* test methodology in other application domains than just the processing of data streams. It is expected, in reaching this main objective, to identify possible adjustments in the *Tricorder* methodology that can improve its automation, effectiveness and efficiency. The methodology used in this project considered four different experiments to reveal defects in cryptography systems, ordering methods, video processing and fluid simulations. The defects analyzed were based on existing defect taxonomies in the literature and, whenever possible, real defects reported by their developers were used. The metrics used in the evaluations considered the ability to reveal defects and the number of false positives and false negatives. The results of the experiments show that *Tricorder* is able to successfully reveal software defects in the analyzed application domains, for the vast majority of defected executions. This work contributes to the software testing activity, in particular to the cost reduction of this activity, as it extends the use of *Tricorder*, a methodology that automatically detects defects in programs, without the need to create new test cases and oracles.

Keywords: Software testing, Monitoring, Grouping, Performance profiles, Unsupervised machine learning.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

CAPÍTULO

1

INTRODUÇÃO

O desenvolvimento de um software é uma tarefa complexa e dependente da habilidade das pessoas envolvidas e das metodologias seguidas. Muitos são os métodos e ferramentas utilizados para garantir a implementação correta de um software, mas ainda existe a possibilidade de enganos serem cometidos e de defeitos serem inseridos no mesmo. A tarefa de detectar defeitos em software é realizada pelos processos de teste de software. Embora não se possa garantir que um software esteja livre de defeitos, os testes fornecem conceitos, técnicas, metodologias e ferramentas, os quais, juntos, tentam proporcionar uma maior probabilidade de um software realizar as ações para que foi projetado e não realize operações indesejadas (MYERS; SANDLER *et al.*, 2004).

O processo de teste de um software é uma atividade complexa e dependente da habilidade dos profissionais que a realiza (DELAMARO; JINO; MALDONADO, 2013). O teste de todas as combinações de entrada e saída são impraticáveis para a grande maioria das aplicações, com métodos e técnicas sendo utilizados para guiar e viabilizar a realização deste processo. Além da complexidade do processo de teste, as habilidades e atitudes do testador são fatores importantes para o sucesso da realização desta tarefa (MYERS; SANDLER *et al.*, 2004).

Assim como a implementação de um software, a etapa de testes é guiada pelos requisitos funcionais e não funcionais do sistema (GLINZ, 2007). Diversas técnicas de teste de software podem ser utilizadas de acordo com cada requisito, onde é possível destacar as técnicas de teste caixa preta e teste caixa branca (MYERS; SANDLER *et al.*, 2004).

O teste caixa preta tem o objetivo de encontrar defeitos em um software ao avaliar as respostas do programa em teste para um conjunto definido de entradas (DELAMARO; JINO; MALDONADO, 2013). Como é indicado pelo seu nome, o programa em teste é considerado como uma caixa preta, sem analisar nenhum ponto de sua estrutura interna. Os dados utilizados para a geração de casos de teste funcionais são baseados nas especificações do software e a escolha destes dados procura cobrir as diferentes situações em que a aplicação não realiza o que

foi projetada para fazer ou realiza algo para que não foi projetada (MYERS; SANDLER *et al.*, 2004).

Testes caixa branca avaliam as estruturas internas do programa, analisando seus componentes e a lógica associados (DELAMARO; JINO; MALDONADO, 2013). Essa técnica de teste busca testar diferentes componentes internos de um programa, como nós (grupos de comandos), arestas (caminhos possíveis entre nós), fluxo de dados, entre outros aspectos (MYERS; SANDLER *et al.*, 2004).

Uma aplicação funcionando corretamente apresenta uma assinatura própria de uso de recursos, como o uso de processador, memória ou entrada e saída. O desempenho do sistema durante a execução de um software pode ser monitorado a partir da observação de métricas relacionadas ao uso de tais recursos. Após uma modificação no software correto, em uma atualização por exemplo, seu perfil de uso de recursos pode sofrer alterações que podem ser detectadas a partir de sua análise.

Ferramentas de monitoração podem ser utilizadas para analisar o uso de recursos de uma aplicação durante um período de execução e contribuir para a geração de perfis de uso de recursos. Esses perfis indicam o comportamento do software executando em uma determinada configuração de máquinas, utilizada com uma carga de trabalho definida. Alterações no uso de recursos em diferentes versões de um mesmo software podem ser utilizadas para identificar a presença de defeitos inseridos na nova versão.

Dados de monitoração de recursos gerados para diferentes versões de um mesmo software, executadas com cargas de trabalho específicas, podem ser utilizados para gerar conhecimento sobre o comportamento esperado desse software. Diferenças nos perfis de desempenho entre essas versões do software podem ser identificadas a partir do uso de métodos de agrupamento e classificação de dados. Perfis de desempenho de diversas execuções da aplicação original definem as características de uma versão considerada correta da aplicação e podem ser utilizadas para a identificação de anomalias em execuções de novas versões desta aplicação. Essas anomalias podem estar relacionadas a defeitos de software e a detecção das mesmas em novas versões de um software, por meio do agrupamento de perfis de uso de recursos, é tratada neste trabalho.

Diversos trabalhos buscam identificar problemas de desempenho em diferentes escopos de sistemas, onde a monitoração de desempenho é utilizada na identificação de regiões de código que possam afetar o seu desempenho (ATTARIYAN; CHOW; FLINN, 2012; JIN *et al.*, 2012; FOURNIER *et al.*, 2019). A caracterização de perfis de uso de recursos é utilizada em alguns trabalhos, aplicada na execução de versões do mesmo programa ou entre programas com o objetivo de identificar anomalias, uma vez que é esperado um uso de recursos do sistema semelhante para aplicações ou métodos com poucas alterações (SHEN *et al.*, 2015; SOUSA *et al.*, 2016).

A análise de dados de desempenho para o teste de um software é realizada em alguns

casos onde os desenvolvedores ou usuários possuem um bom conhecimento sobre o comportamento do sistema e conseguem detectar anomalias do uso de recursos empiricamente. Um método que forneça uma automatização desse processo pode ser usado como uma etapa de teste onde não é necessário um grande conhecimento sobre o código ou um oráculo para relacionar as entradas e saídas da aplicação.

Além da identificação de problemas de desempenho, alguns poucos trabalhos utilizam o perfil de uso de recursos do programa para a identificação de defeitos relacionados a requisitos funcionais, como o trabalho apresentado por Cherkasova *et al.* (2008). O trabalho desenvolvido por Montes (2019a) apresenta a metodologia *Tricorder*, a qual visa detectar defeitos em software ao combinar técnicas de monitoração de uso de recursos e de agrupamento e classificação. A *Tricorder* foi implementada e validada para o teste de aplicações de processamento de *streams* de dados, utilizando em seus experimentos versões mutantes com defeitos inseridos.

A metodologia *Tricorder* realiza a detecção de defeitos na aplicação, realizando a comparação de perfis de uso de recursos de uma versão considerada correta e da versão em teste da aplicação a partir da aplicação de um algoritmo de aprendizado não supervisionado. Os perfis de uso de recursos dessas versões da aplicação são gerados a partir da monitoração do uso de recursos de sua execução com cargas de trabalho reais, com a implementação apresentada por Montes (2019a) utilizando a ferramenta *Thermometer*, desenvolvida pelo autor do trabalho.

Após monitorar a execução da aplicação correta, os dados gerados são agrupados com a ferramenta *DAMICORE* (Sanches; Cardoso; Delbem, 2011) para a formação de perfis de uso de recursos esperados para a aplicação com as cargas de trabalho utilizadas. Os dados da monitoração da aplicação em teste são comparados aos dados da aplicação original pela ferramenta de agrupamento e classificados ou em um dos grupos já existentes ou formando um novo grupo. De uma aplicação em teste correta é esperado que seu perfil de uso de recursos seja classificado nos grupos referentes à aplicação original (a qual supostamente está correta). Caso um novo grupo seja formado apenas por amostras de execução da aplicação em teste, um alerta é gerado indicando potencialmente a presença de um defeito de software.

A validação inicial da metodologia *Tricorder* proposta por Montes (2019a) é restrita a aplicações de processamento de *streams* de dados, indicando a presença de um defeito caso a etapa de agrupamento retorne um novo grupo de amostras de uso de recursos. Não foram considerados outros nichos de aplicação durante a validação da metodologia. Verificar o comportamento da *Tricorder* em diferentes contexto, diferentes do processamento de *streams* de dados, permitiria estender o uso da Tricorder para aplicações não previstas inicialmente no seu projeto. O teste de outros nichos de aplicação também proporcionaria a análise de novos defeitos e o estudo de novas métricas de uso de recursos.

1.1 Objetivos

O principal objetivo deste trabalho é estender a abrangência da metodologia *Tricorder*, proposta por Montes (2019a), considerando sua eficácia em revelar defeitos em outros tipos de aplicações, não apenas para aplicações que processam *streams* de dados. Outros objetivos são investigar novas métricas de uso de recursos, novos defeitos e novas configurações que otimizem os resultados obtidos com o uso da metodologia. Foram realizadas modificações e adaptações na metodologia para o teste de aplicações de nichos diferentes de processamento de *streams* de dados, como criptografia e simulação de fluidos, além do estudo e investigação de algumas limitações da metodologia mencionados no trabalho prévio, como o uso de apenas um defeito real.

1.1.1 Objetivos Específicos

- Compreender o grau de dificuldade de se configurar os diferentes componentes da *Tricorder* para que ela seja capaz de revelar defeitos em diferentes domínios de aplicação.
- Investigar o uso de recursos nas diferentes aplicações estudadas além do uso de CPU, memória e disco, com vistas à revelação de defeitos pela *Tricorder*.
- Investigar o impacto de diferentes tipos de defeitos no desempenho das aplicações testadas.

1.2 Estrutura do texto

Este trabalho está dividido em outros nove capítulos. O Capítulo 2 apresenta trabalhos relacionados ao uso de monitoração de desempenho para teste de software, mostrando o processo e resultados de um levantamento bibliográfico. No Capítulo 3 são introduzidos conceitos básicos sobre os temas importantes para o trabalho, abordando teste de software, avaliação de desempenho e agrupamento de dados. A metodologia *Tricorder* e o trabalho que a apresentou (MONTES, 2019a) são descritos no Capítulo 4, mostrando uma visão sobre o funcionamento da metodologia e os experimentos realizados para sua validação no teste de um nicho de aplicações. No Capítulo 5 é apresentada a metodologia empregada para os experimentos deste trabalho, descrevendo o ambiente de execução, as aplicações testadas e os defeitos utilizados. Os Capítulos 6, 7, 8 e 9 descrevem os experimentos realizados. Para cada experimento são apresentados a aplicação testada, as métricas monitoradas e os defeitos utilizados para a geração das versões mutantes. Também são apresentados os resultados de cada experimento, mostrando os erros e acertos da metodologia ao realizar os testes das aplicações. As conclusões deste projeto é apresentado no Capítulo 10, onde uma análise dos resultados é descrita. Também são abordadas as possíveis ameaças à validade do trabalho, desafios encontrados durante sua execução, principais contribuições dos experimentos realizados e trabalhos futuros.

CAPÍTULO

2

TRABALHOS RELACIONADOS

2.1 Considerações iniciais

Neste capítulo são apresentados trabalhos relacionados aos temas da pesquisa. Serão abordados os trabalhos utilizados como referência por Montes (2019a) e que foram utilizados para realizar um mapeamento sistemático realizado durante o desenvolvimento do presente projeto. Estes trabalhos são utilizados para levantar conhecimento sobre o que tem sido pesquisado em relação à automatização de testes de software, testes de desempenho e a processos de agrupamento, monitoração e análise de dados de uso de recursos.

2.2 Revisão bibliográfica

Uma revisão bibliográficaacerca dos temas da pesquisa foi realizado com a finalidade de encontrar os trabalhos mais relevantes relacionados a esses temas. A partir dos artigos utilizados como referência no trabalho prévio, foram escolhidos seis trabalhos para formar um grupo de controle da busca realizada, com a *string* de busca utilizada sendo formada por termos presentes nesses trabalhos. Foram utilizados os seguintes artigos como grupo de controle para a revisão bibliográficarealizada:

- *Anomaly? Application Change? or Workload Change?* (CHERKASOVA *et al.*, 2008) - Propõe um método para a detecção de anomalias e alterações de desempenho de uma aplicação causadas por alterações realizadas no decorrer de seu ciclo de vida. São analisadas métricas relacionadas ao consumo de CPU para caracterizar o comportamento da aplicação em conjunto com sua assinatura de performance para a identificação de anomalias e das modificações relacionadas à sua manifestação.
- *Using Magpie for request extraction and workload modelling* (BARHAM *et al.*, 2004) - Apresenta uma ferramenta para a caracterização de uma aplicação em relação ao uso de re-

cursos do sistema. Eventos do sistema operacional e outros componentes são relacionados a requisições realizadas pela aplicação para a construção de modelos representativos do comportamento do sistema.

- *Capturing, Indexing, Clustering, and Retrieving System History* (COHEN *et al.*, 2005) - É apresentado um método de extração automática de assinaturas de estados do sistema com o objetivo de agrupar e classificar problemas de execução encontrados durante o histórico da aplicação. Analisar a similaridade das assinaturas do sistema permite a identificação de novos problemas da aplicação e o desenvolvimento de diagnósticos anteriores.

- *Understanding and Detecting Real-World Performance Bugs* (JIN *et al.*, 2012) - Apresenta um estudo sobre problemas de performance em aplicações reais que resulta em um guia para a identificação e prevenção de problemas semelhantes. As regras elaboradas durante o estudo são aplicadas nos programas utilizados, com a detecção de um número considerável de problemas de desempenho desconhecidos.

- *X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software* (ATTARIYAN; CHOW; FLINN, 2012) - Introduz uma técnica para a identificação da causa de problemas de desempenho de uma aplicação. O comportamento da execução de blocos da aplicação são analisados e relacionados aos problemas de desempenho encontrados, com o objetivo de identificar suas causas. Foram realizados experimentos em aplicações reais com resultados positivos para a identificação de problemas e para o impacto da ferramenta no desempenho do sistema.

- *Analysis of Application Performance and Its Change via Representative Application Signatures* (MI *et al.*, 2008) - Apresenta uma abordagem para a definição de assinaturas de desempenho de aplicações baseadas em ambiente de produção, comparando a performance de versões atuais e anteriores do sistema e detectando alterações inseridas em atualizações. São consideradas métricas de latência e execução de transações realizadas para a definição das assinaturas da aplicação.

As ferramentas de busca *IEEE Xplore*, *ACM Digital Library* e *Scopus* foram escolhidas para uso nessa fase do trabalho, onde foi aplicada a *string* de busca e trabalhos encontrados foram analisados. A *string* de busca utilizada é composta por termos referentes a anomalias e assinaturas de desempenho, detecção de defeitos e agrupamento:

("Performance"+ ("Analysis OR "Anomaly"OR "Anomalies"OR "Bugs"OR "Bug"OR "Objectives"OR "Objective)) AND ("Debugging"OR "Debug"OR "Clustering"OR "Application signature")

Pelo alto número de trabalhos não relacionados ao escopo da pesquisa observados após a execução da *string* de busca, foram adicionados termos relacionados à área médica, processamento de imagens e hardware como restrição:

NOT("medic"OR "image"OR "medical"OR "dataset"OR "visualization"OR "hardware")

A restrição de artigos relacionados à área médica e a processamento de imagens foi inserida pela grande quantidade de artigos retornados dessas áreas que utilizam detecção de anomalias ou agrupamento de dados em suas soluções. A análise de dados de desempenho também é utilizada para a detecção de falhas de hardware, justificando a restrição desse termo na busca realizada.

Na primeira fase da busca, após a execução da *string* de busca apresentada anteriormente, foram encontrados 2974 artigos nas três bases, com trabalhos aparecendo em mais de uma base e inflando esse número. Uma nova busca foi realizada adicionando uma restrição de tempo para publicação de até cinco anos da data da pesquisa, de forma a encontrar o estado da arte dos trabalhos das áreas de interesse. O número de trabalhos retornados foi limitado a 907 trabalhos após a restrição de data de publicação, dentre os quais 245 trabalhos foram detectados como duplicatas, restando 662 trabalhos para análise na fase II.

A primeira análise dos resultados da revisão bibliográficafoi realizada com a leitura do título e resumo dos trabalhos, onde 577 trabalhos foram descartados nesta etapa por não apresentarem temas muito próximos às áreas de interesse da pesquisa. Na terceira fase foi realizada a leitura do resumo, introdução e conclusão dos 85 trabalhos restantes, onde 30 trabalhos foram selecionados para a última etapa, onde foi realizada a leitura completa dos artigos e estabelecida uma classificação dos trabalhos quanto a seus objetivos e metodologias. Um resumo da quantidade de trabalhos analisados pelas fases de busca é apresentado na Tabela 1:

Tabela 1 – Fases de seleção de artigos

Fase	Quantidade de artigos selecionados
I	662
II	85
III	30

Após a leitura dos trabalhos selecionados, uma classificação desses trabalhos foi implementada com o objetivo de agrupá-los de acordo com os temas relevantes para esta pesquisa. Foram criados oito grupos de trabalhos, apresentados pela descrição a seguir e pela Tabela 2.

1. Caracterização do desempenho do sistema - Apresentam ferramentas, métodos ou metodologias para descrever o comportamento do sistema em relação ao seu desempenho. Este é

um dos principais pontos deste trabalho, com a possibilidade de se realizar comparações entre os métodos utilizados. Trabalhos como os apresentados por Wen *et al.* (2016) e por Bhattacharyya e Amza (2018) utilizam métodos de caracterização do comportamento do sistema em teste nas ferramentas apresentadas.

2. Perfilamento - Propõem o uso de perfis de desempenho para a descrição do desempenho do sistema, parcialmente ou por completo. Esta é uma estratégia utilizada neste trabalho, onde os trabalhos encontrados poderão reforçar o seu entendimento e fornecer comparações sobre o seu uso. Shen *et al.* (2015) realizam a criação de perfis de desempenho de aplicações web para predizer seu comportamento de acordo com a entrada e Sousa *et al.* (2016) utilizam essa técnica para encontrar problemas de execução em simulações computacionais.

3. Agrupamento de dados de desempenho - Utilizam ferramentas, métodos ou metodologias para agrupar descrições sobre o desempenho de sistemas ou de problemas encontrados para a identificação ou classificação de problemas relacionados a desempenho. Fournier *et al.* (2019) e Sauvanaud *et al.* (2015) utilizam o agrupamento de dados relacionados a desempenho para diferentes arquiteturas: enquanto o primeiro busca defeitos em aplicações web, o segundo analisa os dados de desempenho de *clusters* em nuvem.

4. Causa de anomalias de desempenho - Têm o objetivo de encontrar as regiões do sistema responsáveis por algum problema relacionado a desempenho. Esses trabalhos variam entre a identificação de porções de código que causam anomalias de desempenho e aplicações em sistemas em nuvem, onde problemas de desempenho podem estar relacionados a uma aplicação específica. Os objetivos desses trabalhos diferem do objetivo da metodologia *Tricorder* por buscar apenas problemas de desempenho e não verificar defeitos que alteram a saída da aplicação. Dai *et al.* (2018) e Wen *et al.* (2016) são exemplos de trabalhos que apresentam ferramentas de detecção de problemas de performance, ambos visando aplicações baseadas em servidores.

5. Ferramentas de monitoração de desempenho - Trabalhos que apresentam ou utilizam alguma ferramenta para a monitoração do desempenho do sistema em teste. Podem ser úteis para a comparação da eficiência e impacto da ferramenta de monitoração *Thermometer*, já utilizada na metodologia *Tricorder*. Faasse, Bucek e Schmidt (2020) e Wert, Schulz e Heger (2015) apresentam ferramentas de monitoração de desempenho, avaliando o uso de recurso das aplicações analisadas e buscando diminuir o impacto da ferramenta no desempenho do sistema.

6. Ferramentas e métodos de análise de métricas de desempenho - Trabalhos que apresentam ferramentas, métodos ou metodologias com objetivo de analisar dados sobre o uso de recursos da aplicação em teste. O estudo desses métodos pode auxiliar na evolução da metodologia aplicada neste trabalho, além da comparação das estratégias utilizadas. A

maior parte dos trabalhos analisados utiliza essas ferramenta e métodos, como Kou e Chen (2018), que busca identificar defeitos em aplicações de *big data*, ou Sujon *et al.* (2016), que propõe uma abordagem de detecção de problemas de desempenho ao analisar o código fonte da aplicação em teste.

7. Detecção de anomalias de desempenho - Esses trabalhos têm o objetivo de detectar anomalias de desempenho na execução da aplicação em teste. A detecção de anomalias é parte do objetivo deste trabalho, permitindo a comparação com os métodos utilizados nesses trabalhos e seus resultados. Em Wurzenberger *et al.* (2017) são analisados dados de desempenho de aplicações para a detecção de anomalias no comportamento da aplicação em teste, assim como é realizado em Wang *et al.* (2015).

8. Uso de dados históricos - Utilizam de dados sobre o desempenho de versões anteriores ou de sistemas com características semelhantes para comparação com o atual sistema em teste. Laaber e Leitner (2017) e Mühlbauer, Apel e Siegmund (2019) são exemplos de trabalhos que analisam dados de diferentes versões de um software como meio de identificar problemas de desempenho.

Além da revisão bibliográficaoutros trabalhos merecem atenção por sua relação com teste de software, agrupamento e monitoração de recursos.

Rogstad, Briand e Torkar (2013) apresentam uma abordagem para a seleção de casos de teste de caixa preta, baseando-se nas similaridades entre esses casos de teste. A Distância por Compressão Normalizada (NCD) é utilizada na análise de casos de teste de uma aplicação, onde é realizada uma seleção de casos de teste para otimizar os custos e esforços da etapa de testes.

Um estudo sobre defeitos de software relacionados à performance foi realizado por Chen, Winter e Suri (2019). Esse trabalho analisou mais de 700 defeitos em 13 aplicações de código aberto bastante utilizadas com o objetivo de investigar a frequência e complexidade desses defeitos. Foram observados alguns padrões nesse conjunto de defeitos e descritas suas características. A finalidade desse trabalho é fornecer uma base de dados sobre os problemas de desempenho das aplicações analisadas para o uso em pesquisas sobre a detecção desses defeitos, o que pode ser útil para os estudos realizados neste trabalho.

2.3 Considerações finais

Este capítulo abordou alguns trabalhos relacionados aos temas desta pesquisa. Os resultados de uma revisão bibliográficarealizada foram apresentados, com uma classificação dos artigos analisados em relação aos pontos relacionados a este trabalho. Esses artigos serão úteis para o estudo dos componentes da metodologia estudada neste trabalho, fornecendo uma base de comparação sobre sua utilização e impacto.

Tabela 2 – Classificação dos artigos selecionados na revisão bibliográfica

	Categoria							
Trabalho	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
(LAABER; LEITNER, 2017)	-	-	-	-	X	X	-	X
(BANERJEE; SRIVASTAVA, 2019)	-	-	-	-	X	X	-	-
(WANG; JIN; YU, 2018)	-	-	X	-	-	-	-	-
(WANG *et al.*, 2017)	X	-	X	-	-	-	X	X
(MÜHLBAUER; APEL; SIEGMUND, 2019)	X	-	-	-	-	X	-	X
(SILVA *et al.*, 2018)	X	-	-	-	-	X	-	X
(KHATUYA *et al.*, 2018)	X	-	-	-	-	X	X	X
(WERT; SCHULZ; HEGER, 2015)	-	-	-	-	X	-	-	-
(CASOLA *et al.*, 2017)	-	-	-	-	X	X	-	-
(KOU; CHEN, 2018)	X	-	-	X	-	X	X	-
(Wurzenberger *et al.*, 2017)	-	-	X	-	-	-	X	-
(FOURNIER *et al.*, 2019)	-	-	X	-	-	-	X	-
(SHEN *et al.*, 2015)	X	X	-	-	-	-	X	-
(WIENKE; WREDE, 2016)	-	-	-	X	-	X	X	-
(SUJON *et al.*, 2016)	-	-	-	-	-	X	X	-
(SAUVANAUD *et al.*, 2015)	-	-	X	-	-	X	X	-
(MITRA *et al.*, 2015)	X	-	-	-	-	-	-	-
(ALAM; AHMAD; KO, 2017)	X	X	-	-	-	X	X	-
(WANG *et al.*, 2015)	X	-	X	-	-	X	X	-
(REICHELT; KÜHNE, 2018)	-	-	-	-	-	X	X	X
(DAI *et al.*, 2018)	-	-	-	X	-	X	X	-
(SOUSA *et al.*, 2016)	X	X	-	-	-	X	-	X
(FAASSE; BUCEK; SCHMIDT, 2020)	-	-	-	-	X	X	-	-
(REICHELT; KÜHNE; HASSELBRING, 2019)	-	-	-	-	-	X	X	X
(WEN *et al.*, 2016)	X	-	-	X	X	X	X	-
(DELGADO-PÉREZ *et al.*, 2020)	X	-	-	-	-	X	X	-
(SÁNCHEZ *et al.*, 2018b)	X	-	-	-	-	X	X	-
(BHATTACHARYYA; AMZA, 2018)	X	X	-	-	-	-	X	X
(ARORA *et al.*, 2018)	-	-	-	X	X	X	-	-
(SÁNCHEZ *et al.*, 2018a)	X	-	-	X	X	X	X	-

Diversos trabalhos apresentam ferramentas e métodos para simplificar o processo de teste de software, buscando reduzir o custo e o esforço necessários para realizar essa etapa. Foram encontrados trabalhos que realizam a análise de uso de recursos do sistema para a detecção de anomalias e defeitos relacionados ao desempenho de aplicações de diversos nichos, como aplicações web, baseadas em nuvem e de *big data*. Métodos de mineração de dados foram utilizados nos dados de monitoração em parte dos trabalhos selecionados, assim com o uso de dados de monitoração de diferentes versões da aplicação em teste

A maior parte dos trabalhos analisados investigavam a presença de defeitos de desempenho, sem observar os defeitos relacionados a requisitos funcionais da aplicação ou tinham o desempenho como requisito de software. Não foram encontrados trabalhos que utilizavam a mo-

nitoração de uso de recursos para teste funcional de softwares que não apresentam o desempenho como requisito, mostrando uma lacuna onde se insere a metodologia *Tricorder*.

CAPÍTULO

3

REFERENCIAL TEÓRICO

3.1 Considerações iniciais

Neste capítulo são apresentados brevemente alguns dos principais tópicos relacionados ao tema do trabalho, destacando-se, em sequência, conceitos de teste de software, avaliação de desempenho e agrupamento de dados.

3.2 Teste de software

O processo de desenvolvimento de software é uma tarefa complexa e dependente da habilidade de seus desenvolvedores. Apesar do uso de ferramentas e métodos de engenharia de software, erros podem surgir durante o desenvolvimento (DELAMARO; JINO; MALDONADO, 2013). Testes de software são aplicados com o objetivo de detectar esses erros (MYERS; SANDLER *et al.*, 2004) e garantir a conformidade com a especificação, formando um conjunto de atividades de Teste, Validação e Verificação (TV&V) (DELAMARO; JINO; MALDONADO, 2013).

A execução de um software deve ser consistente e previsível de acordo com sua especificação, com os processos de Teste de Software sendo aplicados para garantir que o sistema desenvolvido faça o que foi projetado para fazer e não realiza nenhuma operação indesejada (MYERS; SANDLER *et al.*, 2004). É papel do testador analisar os requisitos do sistema em teste e projetar os casos de teste com o objetivo de revelar defeitos desconhecidos. Garantir que um software não possui nenhum defeito é uma tarefa impraticável, geralmente impossível (MYERS; SANDLER *et al.*, 2004), contribuindo para o elevado custo do processo de teste de software juntamente com a complexidade de se projetar cada caso de teste.

Softwares são implementados seguindo os requisitos especificados em seu projeto. Esses requisitos podem ser classificados como *Requisitos Funcionais* e *Requisitos Não Funcionais*.

Glinz (2007) separa as definições existentes para *Requisitos Funcionais* em duas linhas de pensamento que concordam bastante. A primeira linha foca na definição de de requisitos funcionais como as funções que o sistema deve realizar, o que o programa deve fazer. A segunda linha foca no comportamento do sistema, como o programa deve realizar as funções definidas. A principal diferença citada por Glinz (2007) entre essas duas linhas de definições sobre requisitos funcionais está relacionada a requisitos de tempo de execução, vistos como características do comportamento do sistema embora não sejam funcionais.

Seguindo sua pesquisa, Glinz (2007) cita uma lista de definições para *Requisitos Não Funcionais*, considerando características, atributos e performance do sistema, entre outros aspectos. São requisitos funcionais aspectos relacionados à qualidade e performance do sistema, além de outras restrições necessárias para alcançar os demais requisitos. Dentre os requisitos não funcionais, é possível citar restrições de tempo e espaço, vazão, usabilidade, segurança, portabilidade e aspectos legais e ambientais.

3.2.1 Técnicas

Como forma de guiar o processo de teste de software, são utilizadas algumas técnicas de teste, onde pode se destacar o Teste Estrutural, o Teste Funcional e o Teste Baseado em Defeitos. Estas técnicas são baseadas em requisitos funcionais.

Teste Funcional é uma técnica de teste de caixa preta, realizado apenas com casos de teste para a entrada do programa e o código da aplicação não é considerado. Essa técnica avalia o software pelo ponto de vista do usuário, destacando os defeitos que afetam a resposta (DELAMARO; JINO; MALDONADO, 2013). É necessário um oráculo para a criação dos casos de teste, onde o testador define as entradas para o programa e as saídas corretas que deveriam ser encontradas.

Dentre as técnicas de teste funcional, é possível destacar as técnicas de particionamento em classes de equivalência e a análise do valor limite (DELAMARO; JINO; MALDONADO, 2013). A técnica de particionamento em classes de equivalência consiste em analisar os valores da saída possíveis para representantes de determinadas classes de entradas, validando a resposta do programa para cada classe. A análise do valor limite avalia o comportamento do sistema em teste para entradas localizadas nos limites das classes de entradas existentes, onde, segundo Myers, Sandler *et al.* (2004), há uma possibilidade maior de encontrar defeitos.

O processo de teste de software não garante a ausência de defeitos em um programa pela impossibilidade de se testar todas as possíveis entradas. Casos de teste são definidos para a aplicação dos testes funcionais e estruturais, buscando utilizar os casos com maior possibilidade de identificar um defeito. São diversas as técnicas que apoiam a definição de casos de teste, como classes as técnicas de particionamento em classes de equivalência e análise do valor limites já citadas, onde um conjunto de casos de testes eficiente sendo definido a partir do uso combinado

dessas e de outras técnicas (MYERS; SANDLER *et al.*, 2004).

O **Teste Estrutural** é uma técnica de teste de caixa branca, com o objetivo de executar determinadas porções do código ou de seus componentes (MYERS; SANDLER *et al.*, 2004). Esta técnica permite avaliar a execução dos possíveis caminhos do software, indicando o uso do código pelos casos de teste utilizados.

O teste baseado em fluxo de controle e baseado em complexidade são alguns dos representantes das técnicas de teste estrutural. O objetivo do teste baseado em fluxo de controle é avaliar a cobertura de comandos e desvios de um programa, enquanto o teste baseado em complexidade avalia a cobertura de caminhos básicos de um programa. Um caminho básico ou independente é aquele que apresenta pelo menos uma nova condição (desvio de fluxo do programa) ou um novo conjunto de comandos a serem executados. A cobertura de caminhos básicos permite executar todos os comandos e desvios possíveis de um programa (DELAMARO; JINO; MALDONADO, 2013).

As técnicas de **Testes Baseados em Defeitos** são aplicadas para demonstrar que um determinado defeito não está presente no programa (MORELL, 1990). É possível destacar a técnica de Teste de Mutação, onde um defeito é inserido no programa com a finalidade de melhorar os casos de teste utilizados em testes funcionais. Os defeitos utilizados em testes de mutação são escolhidos para representar os tipos de defeitos inseridos por programadores, onde os casos de teste já utilizados deveriam detectar os defeitos inseridos (JIA; HARMAN, 2011).

Além dos requisitos funcionais, testes também podem considerar requisitos não funcionais. Esses requisitos são características da aplicação em teste que não alteram a saída apresentada, podendo considerar segurança, desempenho, usabilidade, manutenibilidade, entre outros. É possível destacar o desempenho dentre os requisitos não funcionais por ser mais sensível a modificações realizadas em um software e pela possibilidade de indicar problemas durante sua execução.

Testes de requisitos funcionais são realizados por técnicas bem definidas e efetivas, enquanto o teste de requisitos não funcionais são realizados de acordo com as características e necessidades de cada sistema (METSA; KATARA; MIKKONEN, 2007). Os diferentes requisitos não funcionais necessitam de diferentes formas de realizar os testes, geralmente executados separadamente dos testes funcionais.

Testes de desempenho são executados para prever o comportamento do sistema em ambiente de produção, considerando a visão do usuário, ou aferir o nível de disponibilidade do sistema (VOKOLOS; WEYUKER, 1998). São avaliadas características do funcionamento do sistema em teste como uso de recursos, tempo de resposta e vazão buscando a validação da performance como requisito do sistema.

3.3 Avaliação de desempenho

A avaliação de desempenho consiste no emprego de técnicas utilizadas para obter informações de desempenho de hardware e software podem ser consideradas para atingir os objetivos de seleção de equipamento, projeção do desempenho de um sistema em construção ou para a monitoração de um sistema existente (JR, 1971).

3.3.1 Técnicas

As técnicas de avaliação de desempenho se dividem entre técnicas de modelagem, onde é considerada uma abstração do sistema em teste, e de aferição, que realizam experimentações com o sistema.

As técnicas de **Modelagem** consistem na criação de um modelo matemático analítico do sistema em teste. São utilizadas estratégias como o uso de teoria de filas para representar o funcionamento de um programa e realizar a previsão de carga de trabalho e desempenho esperados. Os modelos criados podem ser alterados facilmente, permitindo a avaliação do impacto de novos parâmetros modificados ou inseridos (JR, 1971).

A **Monitoração** está inserida entre as técnicas de aferição. Essa técnica consiste em analisar métricas relacionadas ao uso de recursos de hardware e software do sistema onde a aplicação em teste executa. A monitoração fornece dados sobre o estado atual do sistema, que podem ser utilizados para realizar uma previsão sobre o impacto de configurações de hardware e software sobre o funcionamento da aplicação e obter um perfil de seu uso de recursos, auxiliando a tomada de decisões estratégicas sobre essa aplicação e seu uso (JR, 1971).

Podem ser utilizadas ferramentas de monitoração baseadas em software ou hardware, com suas vantagens e desvantagens. As ferramentas de software têm a vantagem de apresentar um baixo custo de adesão, apesar de impactar no uso de recursos do sistema e alterar a leitura realizada, mesmo que minimamente. Ferramentas baseadas em hardware não afetam o uso de recursos do sistema, porém necessitam de um investimento maior para seu uso. Neste trabalho serão consideradas as ferramentas de monitoração baseadas em software, cujo funcionamento é baseado em informações fornecidas pelo sistema operacional utilizado.

3.3.2 Métricas de uso de recursos

As ferramentas de monitoração apresentam métricas sobre o consumo dos diferentes recursos computacionais, geralmente com apoio do sistema operacional. Dentre as métricas de consumo de recursos existentes é possível destacar as métricas relacionadas ao uso do processador (CPU), memória principal (RAM) e de dispositivos de entrada e saída, geralmente representados pelos dispositivos de rede e de armazenamento (Disco). Algumas dessas métricas serão descritas a seguir (RODOLA, 2020).

As métricas de uso do *Processador* são relacionadas ao uso de processamento pelo sistema completo ou por um processo específico. É possível citar as porcentagens e tempos de de uso do processador, sua média de carga por um determinado período de tempo e informações relacionadas a processos, como tempos de usuário e de sistema utilizados.

Métricas relacionadas à *Memória Principal* do sistema são representadas pelo seu uso e disponibilidade. Total de memória utilizada e disponível, assim como dados sobre memória de *swap* são informações sobre o uso de memória do sistema e sobre seus processos.

Estatísticas de *Entrada e Saída* podem ser obtidas ao monitorar o uso de rede e armazenamento. Dados sobre a quantidade e tamanho de operações realizadas, assim como quantidade de arquivos e conexões abertas podem fornecer informações sobre o sistema e processos, de onde também é possível identificar seus arquivos e conexões relacionados.

Além das métricas relacionadas diretamente à utilização de recursos de hardware, é possível monitorar informações fornecidas pelo *Sistema Operacional* sobre os processo em execução. Dados sobre trocas de contexto, *threads* em execução e usuários conectados podem indicar informações relevantes sobre o uso de recursos de um processo.

Diversas ferramentas são utilizadas para a monitoração de recursos de sistemas computacionais. É possível citar as ferramentas *Ganglia* (PROJECT, 2020) e *Zabbix* (LLC, 2020) ou bibliotecas como a *psutil* (RODOLA, 2020), que fornecem informações sobre os recursos de hardware e software do sistema.

3.4 Agrupamento de dados

Mineração de dados geralmente é utilizada em áreas com bases de dados volumosas, onde tem o objetivo de reconhecer padrões entre os dados que irão se tornar conhecimento (ZHU; DAVIDSON, 2007). São utilizadas diversas técnicas de mineração de dados, apropriadas para os diferentes problemas e objetivos e para as características de cada base de dados, buscando diminuir a quantidade de informações utilizadas para a geração de conhecimento sobre os dados.

As técnicas de Agrupamento de dados podem ser destacadas dentre as técnicas de mineração de dados. O processo de agrupamento é uma técnica de aprendizado não supervisionado e consiste na divisão dos dados em grupos, possuindo uma maior similaridade entre seus elementos e diferenças quando comparados com elementos de outros grupos (BERKHIN, 2006).

Os métodos de agrupamento dependem de uma métrica determinada para calcular a distância entre os elementos, definindo sua similaridade para a formação dos grupos (CESAR, 2016). A escolha da métrica depende de características dos dados de entrada, como vetores de características, imagens e arquivos de código fonte.

3.4.1 DAMICORE

A metodologia *Damicore* (Data Mining of Code Repositories), apresentada por Sanches, Cardoso e Delbem (2011), combina as técnicas *Normalized Compression Distance* (NCD), *Neighbor Joining* e *Fast Newman algorithm* para a mineração de dados em repositórios de códigos. Uma versão dessa metodologia foi implementada e disponibilizada por Cesar (2016) e será a considerada neste trabalho.

O funcionamento do *Damicore* é iniciado calculando uma Matriz de Distâncias entre os elementos do conjunto de dados analisado, utilizando a Distância de Compressão Normalizada (NCD) a partir da compressão realizada pelo método *ppmd* (DEVELOPERS, 2020). A Matriz de Distâncias gerada é transformada em uma árvore binária sem nó raiz, indicando os elementos mais similares por meio das arestas criadas. O método de detecção de estruturas de comunidades em redes *Fast Newman* (NEWMAN, 2004) é aplicado à árvore binária obtida, resultando em um agrupamento hierárquico dos dados de entrada, objetivo da metodologia (CESAR, 2016).

A métrica *NCD* (CILIBRASI; VITÁNYI, 2005) é baseada na *Complexidade de Kolmogorov*, que indica o comprimento da menor descrição de um dado de entrada em uma linguagem universal fixada. A complexidade de *Kolmogorov* pode ser utilizada para fornecer uma medida de similaridade quando aplicado em um par de elementos representados por *strings*, com a *Distância de Informação Normalizada* (*NID*) utilizando essa medida para uma métrica de similaridade. O cálculo da *NCD* é baseado no cálculo da *NID*, utilizando compressão de dados para calcular um limitante superior para a complexidade de *Kolmogorov*, que é incomputável. A métrica *NCD* não depende de uma estruturação dos dados analisados para seu funcionamento, detectando características de similaridade a partir do resultado obtido pela aplicação de um compressor de arquivos sobre os dados de entrada (CESAR, 2016).

A matriz de distâncias calculada com a *NCD* pode ser representada como um grafo completo, onde os vértices representam os elementos da entrada e as arestas representam as distâncias entre os elementos. Para realizar o agrupamento, a metodologia *Damicore* aplica uma simplificação dos dados da matriz, transformando-a em uma rede complexa com os dados mais significantes. Dentre os possíveis métodos para realizar essa simplificação, onde é possível citar os métodos de Limiarização e *kNN*, foi escolhido o método de *Reconstrução Filogenética*, que constrói uma árvore evolucionária de um grupo a partir da distância entre os seus elementos (CESAR, 2016).

A implementação de uma reconstrução filogenética utilizada no *Damicore* foi a do método *Neighbor Joining* (*NJ*), apresentado por Saitou e Nei (1987). O *NJ* aplica uma abordagem gulosa que une pares de elementos vizinhos mais próximos em um ancestral comum, repetindo esta etapa de forma recursiva conectando todos os nós.

O último passo da metodologia *Damicore* é a detecção de comunidades dentre os elementos da árvore gerada pela reconstrução filogenética. O método utilizado nessa etapa é

o *Fast Newman* (*FN*), apresentado em (NEWMAN, 2004). O funcionamento do *FN* utiliza o conceito de modularidade, relacionado à densidade de arestas de um subgrafo comparado à uma organização dessas arestas de forma aleatória, onde um valor alto indica a formação de uma comunidade. O *FN* é um algoritmo guloso, partindo de comunidades com apenas um elemento e realizando sua junção de forma a aumentar a modularidade do conjunto (CESAR, 2016).

A metodologia *Damicore* foi aplicada em trabalhos de diferentes áreas. O trabalho realizado por Martins *et al.* (2016) utiliza o *Damicore* como ferramenta de agrupamento para guiar a reutilização de otimizações em compiladores, de acordo com a similaridade entre funções do programa. Uma caracterização do perfil de alunos quanto ao uso de sistemas computacionais de educação foi realizado e apresentado por Moro *et al.* (2014), onde o *Damicore* foi aplicado para o agrupamento dos dados obtidos e posterior classificação dos alunos quanto ao seu desempenho.

3.5 Considerações finais

Neste capítulo foram apresentados os conceitos fundamentais para entender o propósito e o funcionamento da metodologia *Tricorder*. A *Tricorder* tem como objetivo ser uma etapa complementar do Teste de Software, utilizando dados de desempenho para a detecção da presença de defeitos relacionados a requisitos funcionais. Também foram utilizados conceitos de teste baseado em defeitos para a validação da metodologia. Os conceitos de avaliação de desempenho são utilizados para entender e monitorar o uso de recursos das aplicações testadas e o impacto das modificações e defeitos inseridos no seu uso de recursos. As análises dos perfis de desempenho serão realizadas a partir de agrupamentos dos dados de uso de recursos das aplicações, onde a presença de uma anomalia pode indicar a existência de um defeito na aplicação.

A união dessas três áreas para o desenvolvimento da metodologia *Tricorder* é apresentada no Capítulo 4, onde são descritas as etapas da metodologia e são apresentados os resultados dos primeiros experimentos realizados com seu uso no teste de aplicações de processamento de *streams* de dados.

CAPÍTULO

4

TRICORDER

4.1 Considerações iniciais

A metodologia *Tricorder* foi proposta no LaSDPC (Laboratório de Sistemas Distribuídos e Programação Concorrente) do ICMC/USP por Montes (2019a) para a identificação de defeitos de software. Montes explorou o impacto desses defeitos no uso de recursos computacionais durante o teste de caixa-preta do software, com resultados positivos para a detecção de defeitos nos experimentos realizados. Este capítulo apresenta resumidamente a metodologia *Tricorder*, destacando suas características, avaliações já desenvolvidas e escopo.

4.2 Metodologia *Tricorder*

A metodologia *Tricorder* tem por objetivo identificar anomalias no uso de recursos computacionais causadas por defeitos de código. Através da criação e comparação de perfis de desempenho de um programa em teste é possível mostrar o padrão de uso de recursos durante execuções da aplicação, possibilitando identificar a presença de tais defeitos. O trabalho apresentado por Montes (2019a) considera como recursos computacionais o uso de processador (CPU), memória principal (RAM) e memória secundária (Disco).

Uma implementação da metodologia de teste de software *Tricorder* foi desenvolvida para sua validação. Perfis do uso de recursos da aplicação foram criados ao realizar uma monitoração periódica durante execuções com cargas de trabalho usuais. Os perfis das execuções de uma versão da aplicação considerada correta são agrupados para determinar o uso esperado de recursos para a execução com diferentes cargas de trabalho definidas. A monitoração da execução da aplicação gera perfis que são comparados com esses grupos, onde uma versão sem defeitos deve, em tese, estar no mesmo grupo dos perfis da aplicação original. Um novo grupo formado apenas por novas execuções da aplicação sinaliza uma anomalia do uso de recursos, o que indica a presença de um possível defeito a ser identificado por testadores e desenvolvedores.

A *Tricorder* foca no teste de aplicações que não apresentam o desempenho como requisito funcional e que estão em ambiente de produção, onde são aplicadas situações reais de carga de trabalho, com um número limitado de entradas e possíveis de serem separadas em categorias. Dentre essas aplicações, as pesquisas já realizadas com a *Tricorder* focaram em aplicações que tratam fluxos contínuos de dados executando de forma contínua e autônoma (MONTES, 2019a).

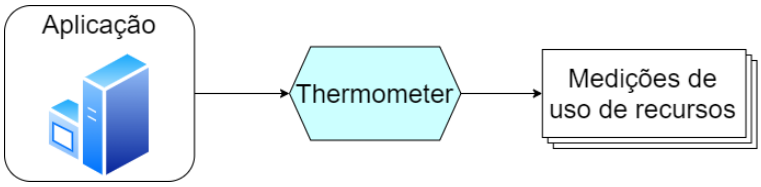
O funcionamento da metodologia *Tricorder* inicia com a monitoração do uso de recursos da aplicação em teste. São monitoradas execuções da versão original da aplicação, considerada sem defeitos, e da versão em teste, onde foi aplicada alguma atualização. Esta etapa é realizada executando a aplicação com as cargas de trabalho determinadas, enquanto a ferramenta *Thermometer* monitora o sistema.

Os arquivos de monitoração obtidos são utilizados na etapa de agrupamento e classificação, utilizando a ferramenta *Damicore*. As informações dos grupos gerados nessa etapa são analisadas para a identificação de anomalias no uso dos recursos pela aplicação em teste. Caso seja identificada uma anomalia no uso de tais recursos, um alerta é produzido para indicar um possível defeito de código na aplicação. Se nenhuma anomalia for detectada durante a etapa de agrupamento e classificação após a quantidade de testes determinada, a execução da aplicação para a carga de trabalho utilizada é considerada como correta.

4.3 Monitoração de recursos

A monitoração dos recursos utilizados pela aplicação em teste observa informações disponibilizadas pelo sistema operacional. São aferidas informações de forma periódica sobre o uso de CPU, memória principal e memória secundária (E/S) pela aplicação.

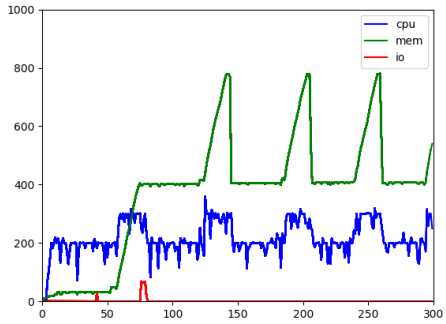
Figura 1 – Funcionamento da ferramenta de monitoração *Thermometer*



Fonte: Elaborada pelo autor.

A ferramenta *Thermometer* (MONTES, 2019c) foi desenvolvida para realizar a monitoração do uso de recursos na metodologia *Tricorder*, com apoio da biblioteca *psutil* (RODOLA, 2020) na linguagem *Python*. A *Thermometer* identifica a execução da aplicação em teste e, como ilustrado na Figura 1, monitora os recursos utilizados durante um intervalo definido e armazena os dados dessa execução em um arquivo *CSV* ("Medições de uso de recursos"). Um gráfico normalizado com os valores obtidos também é gerado e salvo em um arquivo *PNG* para facilitar a análise dos dados de monitoração da aplicação, como é exemplificado na Figura 2. É possível observar as indicações do uso de memória em verde, CPU em azul e Disco em vermelho.

Figura 2 – Exemplo de gráfico de uso de recursos de CPU, RAM e Disco produzido pela *Thermometer*



Fonte: Montes (2019c).

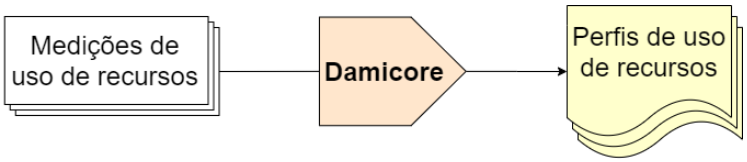
O intervalo de amostragem dessas informações deve ser definido de forma a não interferir de forma significativa no uso de recursos do sistema e, ao mesmo tempo, não deixar de adquirir informações mais precisas sobre eles. Foi escolhido um intervalo fixo de 100 milissegundos entre cada aferição para a amostragem, baseado na documentação da biblioteca *psutil* e sustentado por experimentos preliminares do trabalho (MONTES, 2019a).

Foi escolhido o intervalo de 30 segundos de execução da aplicação em teste para sua monitoração, balanceando o custo do processo da análise dos dados de monitoração e a qualidade da análise realizada. Testes com intervalos de 10 segundos também foram realizados, mas se mostraram insuficientes para a detecção de anomalias. Intervalos de 60 e 120 segundos também foram testados e apresentaram resultados satisfatórios, porém com custo proporcionalmente maior.

4.4 Perfis de uso de recursos

Os perfis de uso de recursos são gerados com um algoritmo de aprendizado não supervisionado, que realiza um processo de agrupamento por similaridade baseado na Distância por Compressão Normalizada (NCD). A ferramenta *Damicore* (Sanches; Cardoso; Delbem, 2011) foi escolhida para realizar esta etapa da metodologia. Os arquivos de medição de uso de recursos são utilizados como entradas para a ferramenta *Damicore*, como ilustrado na Figura 3, que gera os perfis de uso de recursos da aplicação monitorada.

Figura 3 – Geração de perfis de uso de recursos



Fonte: Elaborada pelo autor.

Uma nova execução da aplicação, com uma carga de trabalho já conhecida, pode ser considerada como normal de forma automática após a execução do algoritmo de aprendizado, caso seja classificada em um grupo já existente. O processo de classificação é simplificado por essa etapa, eliminando a necessidade da análise de um especialista. Essa característica também permite a utilização de cargas reais de trabalho, sem a necessidade de geração e análise de entradas artificiais.

A presença de um novo grupo, formado apenas por perfis de uso de recursos da aplicação em teste, resulta em um alerta que indica a possibilidade da presença de um defeito. Essa característica pode ser utilizada para a validação de uma nova versão da aplicação, identificando um consumo de recursos diferente do esperado para uma mesma carga de trabalho.

4.5 Agrupamento

A *Damicore* (Sanches; Cardoso; Delbem, 2011), responsável pela geração e agrupamento dos perfis de desempenho, usa um método de aprendizado não-supervisionado para a identificar anomalias no uso de recursos pela aplicação em teste na metodologia *Tricorder*. A *Damicore* realiza comparações por similaridade entre as amostras, utilizando Distância por Compressão Normalizada (NCD) (Ming Li *et al.*, 2004), para realizar este agrupamento e a classificação dos perfis de uso de recursos.

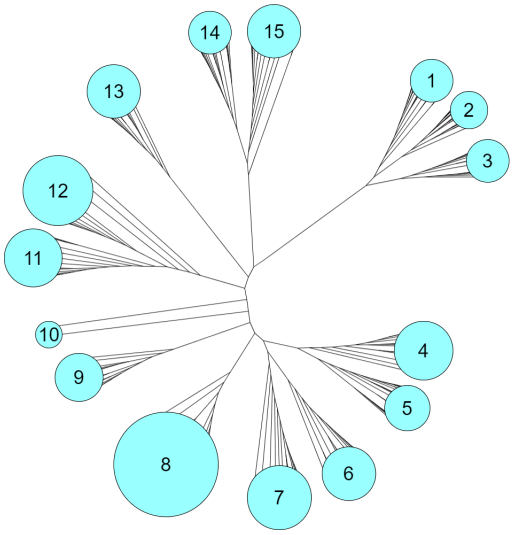
Na implementação da metodologia *Tricorder*, a *Damicore* tem como entrada o diretório com as amostras de uso de recursos da versão original da aplicação. Dentre os possíveis métodos de compressão utilizados pela *Damicore* para calcular a NCD, foram escolhidos os métodos implementados pelas bibliotecas *zlib* (GAILLY; ADLER, 2020) e *ppmd* (DEVELOPERS, 2020).

Os grupos gerados nesta fase representam o uso esperado de recursos para a aplicação para as cargas de trabalho utilizadas. A Figura 4 mostra um agrupamento realizado pela *Damicore*, onde os círculos azuis ilustram possíveis grupos formados por amostras da aplicação original mais similares, a partir da árvore gerada pelos métodos de *Reconstrução Filogenética* e *Neighbor Joining* da *Damicore* (Sanches; Cardoso; Delbem, 2011).

Os arquivos que representam as execuções da versão da aplicação em teste são passados para a *Damicore* de forma incremental. Em cada passo dessa etapa, a amostra de entrada é comparada com os grupos formados anteriormente e classificada, onde a execução de uma versão correta da aplicação deve estar classificada em um grupo formado por execuções da aplicação original.

Após a análise das amostras de entrada, um novo grupo formado apenas pelas novas execuções analisadas indica a presença de uma anomalia no uso de recursos, podendo ser causada por um defeito no código. Um exemplo de anomalia detectada pelo *Tricorder* é apresentado na Figura 5. Como na Figura 4, os círculos azuis ilustram os grupos gerados pelas amostras

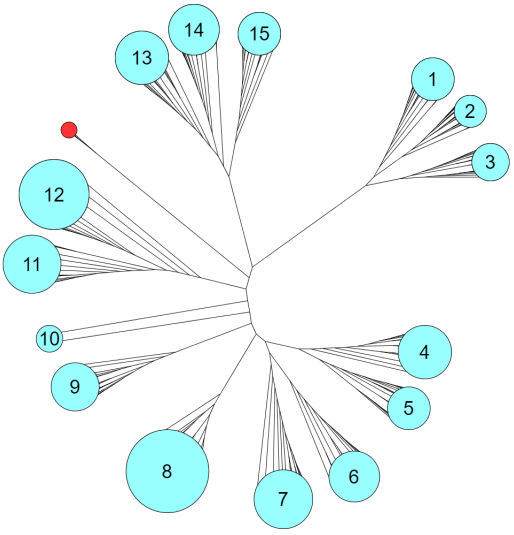
Figura 4 – Exemplo de agrupamento gerado pelo *Damicore*



Fonte: Elaborada pelo autor.

da aplicação original e o círculo vermelho indicando um grupo que representa uma anomalia detectada pela *Tricorder*. Outras formas de analisar o agrupamento gerado poderiam ter sido utilizadas para a indicação de uma anomalia no uso de recursos como, por exemplo, a detecção de amostras diferentes do esperado ainda dentro de um grupo existente. No entanto, inicialmente, a formação de novos grupos com as anomalias foi a maneira escolhida (MONTES, 2019a).

Figura 5 – Exemplo de agrupamento gerado pelo *Damicore*, com anomalia detectada



Fonte: Elaborada pelo autor.

4.6 Validação da metodologia *Tricorder*

A validação da metodologia *Tricorder* teve como objetivo principal verificar a eficácia da mesma em revelar defeitos (MONTES, 2019a) em aplicações de processamento de *streams* de dados.

Para tanto, foram realizados dois experimentos distintos. O primeiro experimento considerou uma aplicação baseada em sistemas de telemetria de dados remotos na arquitetura cliente-servidor, implementada pelo próprio autor (MONTES, 2019a). Nesta aplicação, um processo cliente produz dados que são enviados de forma contínua para um servidor, por meio de comunicação TCP/IP, onde é utilizada uma biblioteca externa para o processamento desses dados. Neste primeiro experimento, defeitos foram inseridos na aplicação em teste utilizando um método semelhante ao utilizado em testes de mutação. As versões da aplicação com defeitos inseridos foram testadas para verificar se tais defeitos poderiam ser revelados com alterações do uso de recursos da plataforma.

O segundo experimento considerou uma aplicação em desenvolvimento em uma indústria brasileira, que realiza cópia de arquivos por rede *Ethernet* (MONTES, 2019a). Nesse segundo experimento a *Tricorder* foi utilizada para detectar um defeito real produzido por um engano da equipe de desenvolvimento do software.

Os dois experimentos são descritos nas próximas seções, sendo que o primeiro deles é apresentado com mais detalhes e o segundo mais resumidamente. Estes níveis de detalhamento acompanham as descrições já apresentadas por Montes (2019a).

4.6.1 O primeiro experimento

O primeiro experimento foi realizado com as ferramentas apresentadas, com a execução do programa original e dos mutantes (que representam versões do programa original com defeito inserido (MONTES, 2019a). Para cada programa gerado, original e mutantes, foram realizadas 30 execuções com cada uma das 15 categorias de cargas de trabalho definidas, de forma a simular a execução em um ambiente real. As execuções foram monitoradas pela ferramenta *Thermometer* pelo tempo definido de 30 segundos, armazenando os dados obtidos em um diretório para uso na fase de agrupamento e classificação.

Foi criada uma base de referência do uso de recursos do programa original, formada por 30 execuções de cada uma das categorias de cargas de trabalho utilizadas. As execuções de cada mutante e para cada carga de trabalho são comparadas separadamente à base de referência para a detecção das possíveis anomalias causadas pelos defeitos inseridos.

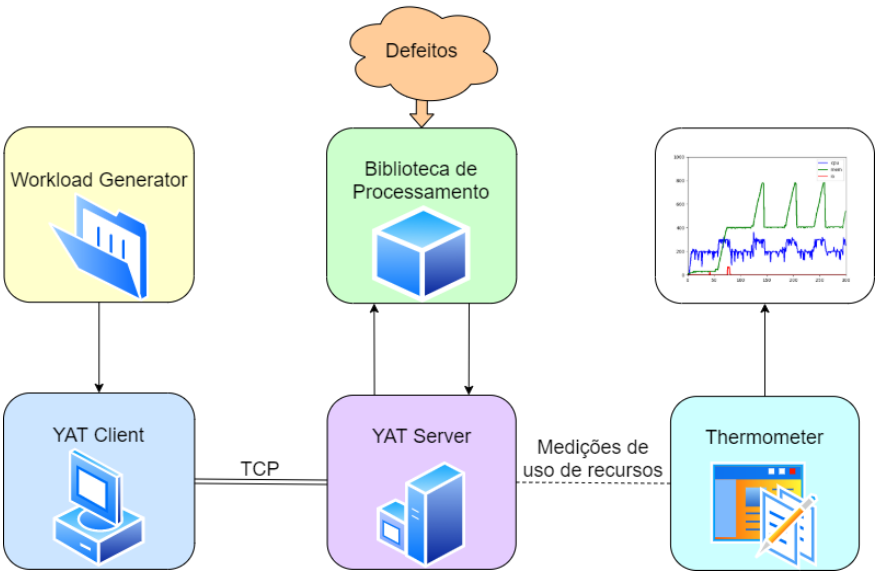
Cada uma das 30 execuções de uma versão mutante é inserida no conjunto de execuções da base de referência de forma incremental, com a ferramenta *Damicore* executando a etapa de agrupamento e classificação. O processo de classificação é repetido para todas as amostras de

execuções da versão em teste, podendo ser interrompido caso uma anomalia seja detectada.

4.6.1.1 Aplicação utilizada

As aplicações **YATServer** (*Yet Another Telemetry Server*) e **YATClient** (*Yet Another Telemetry Client*) foram desenvolvidas para o primeiro experimento de validação da metodologia. Ambas as aplicações foram implementadas em *C++* com o *framework Qt* (QT, 2020) e disponibilizadas por Montes (2019b). Uma representação da arquitetura do sistema é feita na Figura 6 e seu funcionamento é descrito abaixo.

Figura 6 – Arquitetura do sistema desenvolvido para teste



Fonte: Elaborada pelo autor.

Uma ferramenta chamada *Workload Generator* foi desenvolvida para a geração de dados para teste. Essa ferramenta produz arquivos de entrada de acordo com um arquivo de configuração passado. O mesmo arquivo de configuração é utilizado pelo *YATServer*, onde os arquivos de entrada simulam uma leitura de dados e as informações produzidas são enviadas de forma contínua para o servidor em uma conexão TCP.

O *YATServer* aguarda a conexão do cliente para receber os dados de acordo com a configuração utilizada. Uma biblioteca externa, desenvolvida em C++, processa os dados recebidos do servidor em formato *BIN* e *CSV*. Esta biblioteca é o foco principal dos testes realizados, onde são inseridos defeitos em seu código fonte para a detecção pela *Tricorder* (MONTES, 2019a). A saída da aplicação é definida como a primeira janela de dados processada e é salva com o mesmo formato utilizado pela entrada.

A aplicação utilizada para os testes é configurável, permitindo a seleção de tipo de entrada, frequência de envio e dimensão das informações enviadas. Essa configuração é realizada

no início da execução da aplicação e pode alterar o uso de recursos do sistema dependendo da máquina utilizada e seus recursos.

4.6.1.2 Cargas de trabalho

As entradas geradas pela ferramenta *Workload Generator* para a aplicação *YATClient* têm 15 diferentes categorias, onde 6 dessas categorias são configuradas para utilizar apenas um dos formatos *BIN* ou *CSV* e as demais 9 categorias formam entradas mistas, *BIN* e *CSV*. Para cada categoria de entrada simples (*BIN* ou *CSV*) foram definidas cargas de trabalho leve, média e alta, representando uma categoria para cada; enquanto foram definidas três categorias de carga de trabalho leve, média e alta para entradas de tipos de arquivos mistos, como representado na Tabela 3. O comportamento do processamento de cada categoria de entrada impacta de forma diferente no uso de recursos do sistema e pode ou não mostrar a presença do defeito inserido no código fonte. As entradas são geradas aleatoriamente para cada experimento de acordo com sua categoria.

Tabela 3 – Categorias de cargas de trabalho definidas

	Formato de arquivo		
Carga de trabalho	**BIN**	**CSV**	**BIN e CSV**
Low	1	1	3
Medium	1	1	3
High	1	1	3

4.6.1.3 Defeitos inseridos

Com o objetivo de validar eficiência da metodologia em encontrar defeitos de software, foram inseridos defeitos no código fonte da biblioteca de processamento de dados inspirando-se na técnica de teste de mutação. Foram definidas 7 categorias de defeitos, baseadas em problemas reais, que impactam o uso de recursos do sistema sem alterar a saída esperada (MONTES, 2019a). As regiões de código com os defeitos são ativadas durante a compilação, resultando em um programa mutante defeituoso para cada categoria.

As categorias de defeitos definidas no trabalho são:

- **BCLEAN**: Representa defeitos relacionados ao uso excessivo de memória, com o defeito simulando um programa onde uma porção da memória principal foi alocada mas não foi liberada;

- **INFINITE**: Representa defeitos relacionados a processamento desnecessário, com o defeito simulando a espera ocupada por um evento passado ou uma chamada de função com um *loop* sem parada determinada;

- **MONO**: Representa defeitos relacionados a configurações de paralelismo, exemplificado como uma função *multithread* que passa a ser executada como *singlethread*;

- **SLEEP**: Representa defeitos relacionados a chamadas bloqueantes, onde o programa entra em suspensão esperando o resultado dessa chamada. Uma espera não ocupada foi implementada com um comando *sleep*, bloqueando a execução do programa por um determinado tempo;

- **SWAP**: Representa defeitos relacionados a estruturas erradas de pacotes, onde a estrutura enviada é diferente da estrutura que se espera receber. Um exemplo desta categoria de defeitos é a troca de atributos na estrutura de um pacote, a qual pode passar despercebida em testes iniciais devido a possuírem os mesmos tipos de dados e faixas de valores semelhantes;

- **UNLOCK**: Representa defeitos relacionados à sincronização de *threads*, com a liberação indesejada de um *mutex*;

- **NOBREAK**: Representa defeitos de programas que executam de forma diferente para cada tipo de entrada, onde um engano é cometido e o programa executa da forma esperada e de forma errada para uma mesma entrada. Na implementação realizada, a entrada é processada em mais de uma opção de um bloco *switch* onde um comando *break* foi removido.

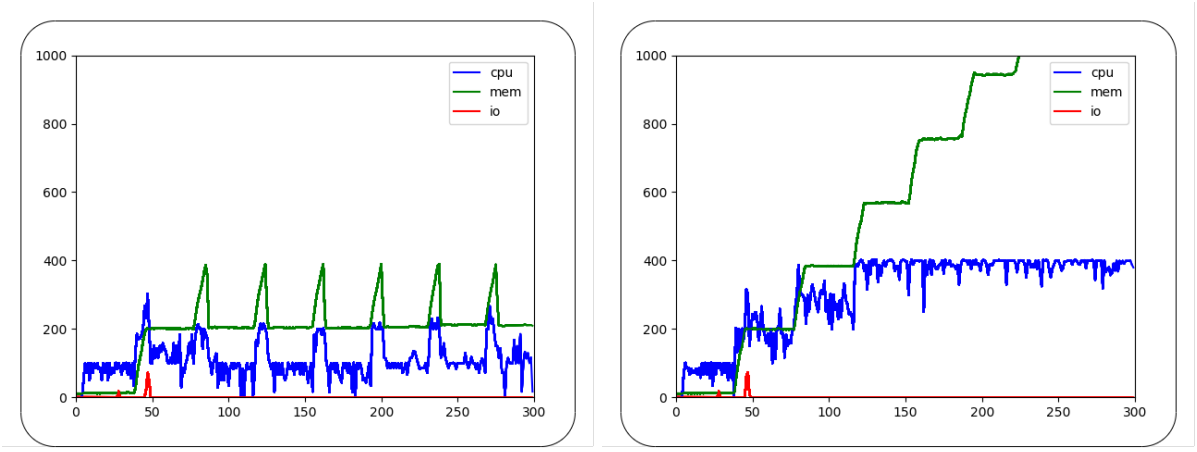
4.6.1.4 Resultados

As descrições da *Tricorder* possuem gráficos relacionados à monitoração de uso de recursos de cada combinação de versão do programa e carga de trabalho, com comparações entre as execuções da versão original e dos mutantes testados (MONTES, 2019a). A Figura 40 ilustra as diferenças de uso de recursos durante a execução do programa correto e de sua versão mutante com o defeito *infinite* para a mesma carga de trabalho, onde é possível observar uma grande diferença nos usos de CPU e memória pelo mutante.

De uma forma geral, a metodologia *Tricorder* foi capaz de identificar a presença de todos os defeitos inseridos nas versões mutantes nos experimentos realizados com cargas de trabalho de único tipo de arquivo, necessitando de pelo menos 3 e no máximo 29 execuções analisadas das 30 execuções monitoradas para cada mutante e carga de trabalho. Além dos defeitos inseridos, foram identificados três casos de falso-positivo, onde foram identificadas anomalias, apesar da aplicação não executar o defeito com o tipo de arquivo de entrada utilizado.

Esses resultados são apresentados na Tabela 4, onde os valores representam o número de execuções necessárias para a detecção das anomalias, com as células em amarelo indicando entradas em que a aplicação executou o defeito do mutante e uma anomalia foi detectada, as células em azul indicando entradas em que a aplicação não executou os defeitos e as células em

Figura 7 – Execuções da aplicação original e aplicação com mutante *infinite*



(a) Aplicação original (b) Mutante *infinite*

Fonte: Montes (2019c).

vermelho indicando os falso-positivos, onde a aplicação não executou mas foi detectada uma anomalia. Nesse experimento foram utilizadas entradas simples, com apenas um formato de arquivo *BIN* ou *CSV*, indicado pela primeira letra da categoria da entrada.

Tabela 4 – Resultados do experimento com entrada simples - ou *BIN* ou *CSV*

	Mutante						
Entrada	**BCLEAN**	**MONO**	**SLEEP**	**SWAP**	**INFINITE**	**NOBREAK**	**UNLOCK**
BL1	9	20	16	21	-	-	-
BM1	6	5	9	9	-	[17]	-
BH1	24	5	15	29	-	-	-
CL1	[15]	-	-	-	6	15	10
CM1	[20]	-	-	-	5	15	6
CH1	-	-	-	-	5	4	3

Fonte: Montes (2019a).

Para cargas de trabalho de tipos de arquivo mistos, onde são utilizados arquivos *BIN* e *CSV*, a metodologia identificou a presença de mais de 80% dos defeitos inseridos nos mutantes. As execuções realizadas com essas entradas sempre ativam as regiões de código com defeito. Foi necessário analisar entre 3 e 26 execuções para a identificação das anomalias, contudo, 12 dos experimentos não identificaram as anomalias com o limite de 30 amostras.

Os resultados encontrados são apresentados na Tabela 5, com os valores novamente representando o número de execuções necessárias para a detecção das anomalias e as células com a letra "N"representando os experimentos onde não foram detectadas as anomalias.

As amostras dos experimentos onde o defeitos não foi detectado apresentavam perfis de uso de recursos semelhantes a perfis da aplicação original executada usualmente com outras

Tabela 5 – Resultados do experimento com entrada mista - *BIN* e *CSV*

	Mutante						
Entrada	**BCLEAN**	**MONO**	**SLEEP**	**SWAP**	**INFINITE**	**NOBREAK**	**UNLOCK**
BCL1	9	12	15	N	6	N	8
BCL2	4	5	17	N	5	N	14
BCL3	13	7	10	14	9	N	N
BCM1	8	9	6	15	7	N	4
BCM2	24	9	8	13	5	N	12
BCM3	8	7	12	22	5	N	13
BCH1	5	4	7	15	5	N	6
BCH2	6	9	7	26	5	N	3
BCH3	16	6	18	N	5	19	7

Fonte: Montes (2019a).

cargas de trabalho. A Figura 8 ilustra essa semelhança, onde (a), (b) e (c) mostram resultados das execuções distintas e (d) mostra resultados das execuções da versão com o defeito *NOBREAK*, todas utilizando a mesma carga de trabalho *BCL2*.

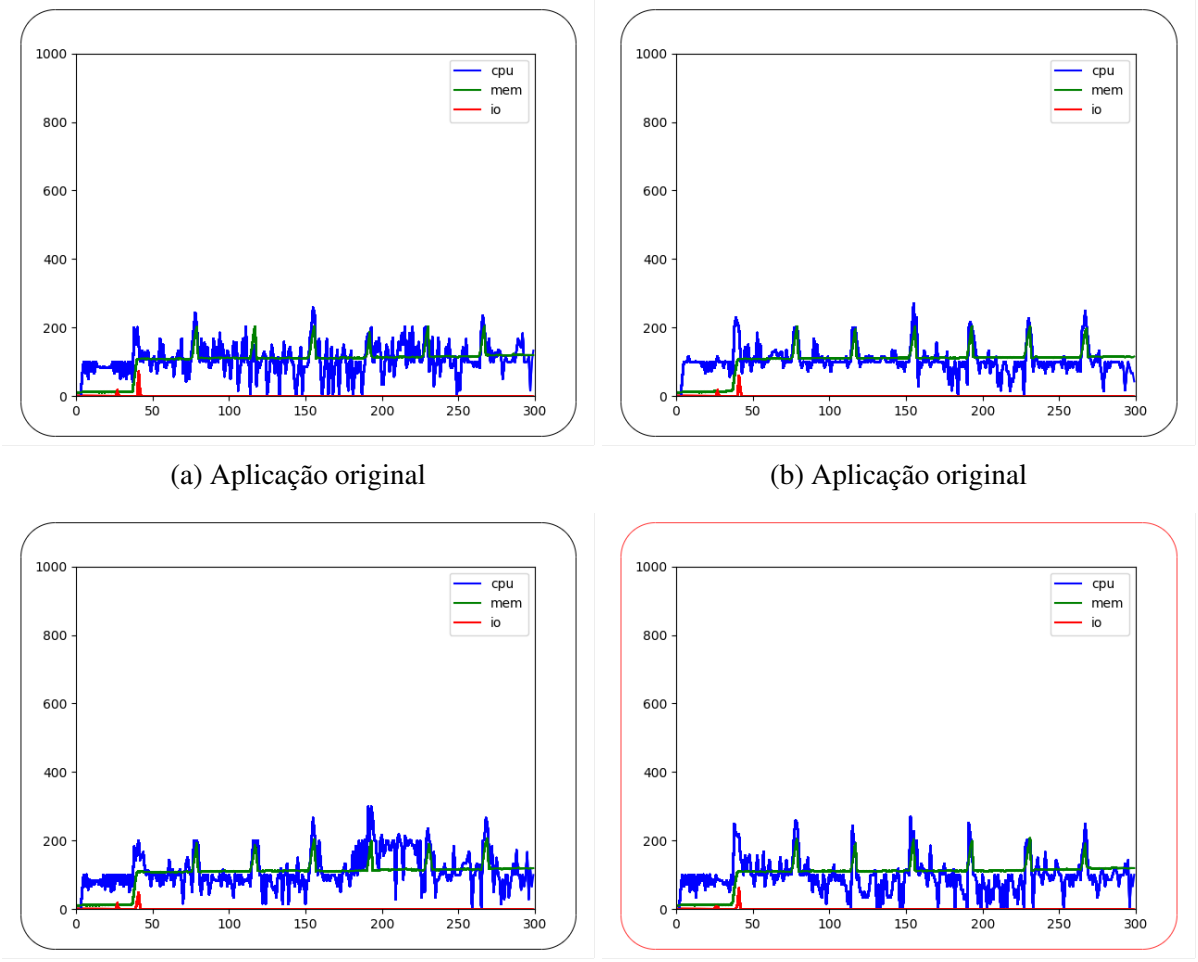
Os estudos realizados com a *Tricorder* chegaram a implementar e testar uma modificação da metodologia, cujo objetivo foi diminuir a ocorrência de falso-positivos (MONTES, 2019a). Essa versão modificada da metodologia considera a presença de defeito somente se anomalias sejam detectadas em duas execuções em sequência. Os mesmos experimentos foram realizados com a versão modificada. Para entradas simples, utilizando arquivos *BIN* ou *CSV*, o número de falso-positivos reduziu para apenas um caso com a entrada *CL1*, porém também ocorreu um caso de não detecção das anomalias para a entrada *BH1*. Quatro novos casos de não funcionamentos foram observado para entradas mistas, que utilizam arquivos *BIN* e *CSV*, onde a presença do defeito *SWAP* não foi detectada para as entradas *BCM1*, *BCM2*, *BCM3* e *BCH1* além das entradas *BCL1*, *BCL2* e *BCH3* que a primeira versão da metodologia também não detectou a presença do defeito (MONTES, 2019a).

O tempo de execução necessário para realizar a etapa de agrupamento e classificação da metodologia *Tricorder* pela ferramenta *Damicore* variou entre 120 e 180 segundos. O máximo de 3 minutos observado nos experimentos foi considerado como aceitável pelo autor, considerando um posterior uso em ambiente de produção (MONTES, 2019a).

4.6.2 Segundo experimento: uma aplicação real

Para o segundo experimento descrito por Montes (2019a), um aplicativo em desenvolvimento de uma indústria brasileira, que realiza cópia de arquivos por rede *Ethernet*, foi utilizado para auxiliar a validação da metodologia *Tricorder* (MONTES, 2019a). A etapa de testes desse aplicativo, realizada pela empresa, identificou um defeito manualmente e o corrigiu. A metodologia *Tricorder* foi aplicada nesse caso real, com o objetivo de avaliar a possibilidade deste defeito

Figura 8 – Exemplo de medições de uso de recursos da aplicação original (a,b,c) e mutante (d) para a mesma carga de trabalho



(a) Aplicação original

(b) Aplicação original

(c) Aplicação original

(d) Mutante *nobreak*

ser identificado automaticamente.

Os experimentos realizados com essa aplicação utilizaram cargas de trabalho reais formadas por cinco conjuntos de dados, com arquivos variando entre 1,47 e 5,49 GB transferidos e processados. Como nos experimentos já apresentados, 30 execuções das versões com e sem defeito da aplicação formaram os dados de entrada para a etapa de agrupamento e classificação.

A presença do defeito foi detectada em todas as execuções com as cinco entradas utilizadas, necessitando de 8 a 28 amostras analisadas. Para a versão corrigida da aplicação, a metodologia detectou apenas um falso-positivo para execuções com as cinco entradas utilizadas. Esse experimento contribuiu para a validação da metodologia em ambiente real de execução, com a utilização de aplicação, defeito e cargas de trabalhos reais para o teste da metodologia (MONTES, 2019a).

4.7 Limitações e Fatores que Afetam a Eficácia da Metodologia Tricorder

O método de agrupamento e classificação por aprendizado não-supervisionado utilizado na metodologia *Tricorder* apresenta a necessidade de algumas amostras para cumprir seu objetivo, detectando uma anomalia no comportamento do programa em teste. Foi necessária a análise de apenas 3 amostras nos melhores casos e de até 29 amostras nos piores caso onde a anomalia foi identificada, resultando em uma média de mais de 10 amostras analisadas para a identificação. Além dos resultados positivos, houve casos onde a análise de 30 amostras não foi suficiente para a identificação de uma anomalia. O tempo necessário para essa análise pode ser um fator limitante para o teste de algumas aplicações.

A ferramenta *Thermometer* precisa ser executada antes da aplicação, utilizando seus primeiros 30 segundos de execução. Apesar de alguns testes preliminares realizados com execuções de 10, 60 e 120 segundos, esses padrões de execução não foram estudados, assim como testes com execuções completas.

Formatos de arquivos de medições de uso de recursos utilizados em aplicações de ciência de dados, como o *Hierarchical Data Format* (HDF5), foram testados para a implementação desenvolvida. A estrutura interna desses formatos não permitiu que a ferramenta *Damicore* funcionasse da forma desejada para os casos analisados. No entanto, estes resultados são preliminares e não foram conduzidos estudos mais profundos neste sentido.

A metodologia *Tricorder* se mostrou eficiente para a detecção de defeitos que afetam o uso de recursos em aplicações que processam *streams* de dados. É esperado que outros tipos de aplicações também possam ser testadas com a metodologia proposta, porém não foram realizados experimentos com outras aplicações. Este trabalho estende esta validação, verificando a eficácia da metodologia *Tricorder* para o teste de diferentes tipos de aplicações, onde o comportamento da metodologia é desconhecido. As características de cada nicho de aplicação impactam o uso de recursos do sistema de acordo com sua necessidade e podem afetar de forma diferente o funcionamento da *Tricorder*.

A ferramenta *Damicore*, utilizada nas etapas de agrupamento e classificação, possui atributos configuráveis, como os algoritmos de simplificação e detecção de comunidades utilizados (CESAR, 2016), que não foram explorados no trabalho prévio. Dentre esses atributos, é possível destacar os algoritmos de compressão utilizados para o cálculo da matriz de distâncias. Os algoritmos de compressão podem alterar o tempo de execução do *Damicore* e as distâncias entre as amostras computadas.

Assim como o algoritmo de agrupamento e classificação utilizado, as métricas de uso de recursos monitoradas impactam diretamente nessa etapa da metodologia. Além disso, essas métricas são responsáveis por realçar as possíveis anomalias causadas pelos defeitos das aplicações

testadas.

4.8 Considerações finais

Neste capítulo foram descrita a metodologia *Tricorder* (MONTES, 2019a). Foi abordado o funcionamento das ferramentas *Thermometer* e *Damicore* e sua participação na metodologia *Tricorder*, durante as fases de monitoração e de agrupamento.

A metodologia *Tricorder* apresentou bons resultados para o teste de aplicações de processamento de *streams* de dados e também para a detecção de defeito real de produção em uma aplicação em desenvolvimento (MONTES, 2019a). A ferramenta *Thermometer* se mostrou eficiente para a monitoração das métricas escolhidas assim como a *Damicore* para a etapa de agrupamento e classificação. A *Tricorder* detectou a maior parte dos defeitos inseridos no primeiro experimento, com poucos casos de falso-positivos.

Algumas limitações podem ser destacadas no processo de validação da metodologia *Tricorder*. O principal ponto foi, basicamente, o uso de apenas um nicho de aplicação em seus experimentos, onde as aplicações desenvolvidas para teste e a aplicação real utilizada realizam processamento de *streams* de dados. O emprego da metodologia com outros nichos de aplicação poderia indicar uma maior abrangência de sua eficácia e possibilitar seu uso para o teste de aplicações nesses nichos. Também é possível destacar a forma que o agrupamento gerado pela *Damicore* foi utilizado, com a presença de uma anomalia sendo detectada apenas quando um novo grupo é formado por amostras da aplicação em teste. Um defeito que faça a execução da aplicação com uma determinada carga de trabalho ser classificada, por exemplo, em um grupo já existente, mas relacionado a execuções com cargas de trabalho diferentes, não é considerada como uma anomalia.

Este atual projeto tem como foco contribuir para diminuir a carência de validações da *Tricorder*, as quais demonstrem com mais profundidade e abrangência virtudes e limitações da metodologia. Como apresentado no próximo Capítulo, pretende-se utilizar a *Tricorder* para o teste de aplicações de nichos diferentes de processamento de *streams* de dados e estudar a forma que são utilizados a *Damicore* e os agrupamentos gerados.

CAPÍTULO

5

METODOLOGIA

5.1 Considerações iniciais

Neste capítulo é apresentada a metodologia utilizada para a realização dos experimentos. Para que os experimentos planejados pudessem avaliar a capacidade da *Tricorder* em revelar defeitos, primeiramente foram levantadas diferentes taxonomias de defeitos de software já existentes na literatura. Essas taxonomias forneceram parâmetros para a escolha dos defeitos a serem inseridos nos códigos, os quais precisam ser detectados posteriormente pela *Tricorder*. Portanto, a segunda Seção deste capítulo descreve as taxonomias de defeitos encontradas na literatura e como elas foram consideradas durante a escolha dos defeitos a serem inseridos nos códigos testados. A seguir, na terceira Seção, o ambiente de trabalho e as configurações utilizadas para a execução dos experimentos são apresentadas, destacando os principais parâmetros, e a etapa de agrupamento e análise. Na quarta Seção são apresentadas as aplicações escolhidas para o teste da metodologia *Tricorder*, com os passos seguidos para a escolha de cada nicho de aplicação estudado. Já no fim deste capítulo, na quinta Seção, são apresentadas as diferentes versões das aplicações geradas, com a contribuição de cada mutante para os experimentos. O Capítulo é encerrado com as considerações finais.

5.2 Taxonomias de defeitos

O estudo apresentado por Avizienis *et al.* (2004) define conceitos relacionados à confiabilidade e segurança em sistemas de computação e comunicação, incluindo conceitos como disponibilidade, segurança, integridade e manutenibilidade, entre outros. O estudo também mostra as ameaças à confiabilidade e segurança, como defeitos, erros ou falhas e seus atributos, além de alguns meios para contornar essas ameaças. Ao descrever os defeitos apresentados, é definida uma taxonomia para a classificação a partir de oito pontos básicos. São definidas oito classes elementares de onde cada defeito pode ser classificado, de acordo com a fase de

ocorrência, limites do sistema, causa fenomenológica, dimensão, objetivo, intenção, capacidade e persistência do defeito, como apresentado na Tabela 6. Das 256 combinações entre essas classes elementares, foram identificadas 31 combinações possíveis, as quais foram separadas em três agrupamentos parcialmente sobrepostos: defeitos de desenvolvimento, físicos e de interação.

Tabela 6 – Representação das classes elementares de defeitos apresentadas por Avizienis *et al.* (2004)

Defeitos	Fase de Criação ou Ocorrência	Desenvolvimento
		Operacional
	Limites do Sistema	Interno
		Externo
	Causa Fenomenológica	Natural
		Feito pelo Homem
	Dimensão	Hardware
		Software
	Objetivo	Malicioso
		Não Malicioso
	Intenção	Deliberado
		Não Deliberado
	Capacidade	Acidental
		Incompetência
	Persistência	Permanente
		Transiente

A partir dessa taxonomia é possível definir todos os defeitos utilizados neste trabalho, como defeitos de desenvolvimento. Das classes elementares, nossos defeitos se categorizam como defeitos da fase de desenvolvimento, com limite interno, causados por humanos, da dimensão de software, não maliciosos e de persistentes. É possível discutir se alguns defeitos, como o *Mono* e o *Sleep*, são inseridos de forma deliberada, por exemplo ao limitar a execução a um único núcleo para determinado teste e não desfazer essa modificação. Na categoria relacionada à capacidade é difícil definir se um defeito foi causado de forma acidental ou por incompetência de um desenvolvedor, mas chegou-se a um consenso de classificá-los como defeitos acidentais. Apesar de classificar todos os defeitos utilizados em basicamente uma categoria, o uso dessa taxonomia é importante para definir de forma clara os defeitos que são alvo da metodologia *Tricorder*.

Hummer *et al.* (2012) apresenta um estudo de potenciais origens e efeitos de defeitos em sistemas distribuídos baseados em eventos. Foi realizada uma derivação de taxonomias da literatura, como o artigo apresentado por Avizienis *et al.* (2004), descrito anteriormente, de forma a criar um modelo que capture as especificidades das subáreas de sistemas distribuídos baseados em eventos. Hummer et al. também elaboraram uma taxonomia baseada no modelo gerado, aplicável a vários tipos de defeito. Nessa taxonomia, os defeitos são classificados de acordo com sua fonte causadora, onde são consideradas como fontes de defeitos o ambiente, entradas externas, funções de código, estado do sistema, componentes de software, conceitos e

abstrações. Os defeitos utilizados neste trabalho são classificados tendo como fonte funções de código, relacionando esses defeitos a problemas de lógica de processamento codificada.

O artigo de Humbatova *et al.* (2020) propõe uma taxonomia de defeitos focada em sistemas de *Deep Learning*. Para a elaboração da taxonomia foi realizada uma análise manual de 1059 artefatos escolhidos de *commits* e *issues* de projetos do GitHub que usam os *frameworks* de *Deep Learning* mais populares. Além dos próprios autores, a taxonomia foi desenvolvida com a participação de outros pesquisadores e desenvolvedores. A taxonomia elaborada classifica os defeitos em cinco categorias: *Model*, *Tensors & Input*, Training, *GPU Usage* e *API*. Na categoria *Model* estão os defeitos relacionados a estruturas e propriedades de um modelo de *Deep Learning*. A categoria *Tensors & Inputs* considera erros de forma, tipo ou formato de dados e a categoria *Training* considera problemas relacionados ao processo de treinamento. A categoria *GPU Usage* considera problemas relacionados ao uso de GPU ao trabalhar com esse nicho de aplicação e a categoria *API* considera defeitos relacionado a algum uso equivocado de Interfaces de Aplicação. Essas categorias abrangem características mais observadas em aplicações de *Deep Learning*, mas muitos dos defeitos são mais genéricos e podem ser observados em outros nichos de aplicações.

A partir da análise dessas taxonomias foi desenvolvida uma taxonomia simplificada, apresentada na Tabela 7, para classificar os defeitos que já foram utilizados em experimentos prévios (MONTES, 2019a) e para auxiliar a definição de novos defeitos para os experimentos deste trabalho. O foco aqui foi nas características de desempenho afetadas pelos defeitos e em categorias das taxonomias estudadas na literatura, afim de convergir para cinco categorias que julgou-se adequadas ao trabalho.

Tabela 7 – Classificação dos defeitos utilizados neste trabalho

Uso de CPU	Uso de GPU	Uso de Memória	Sincronização	Parâmetros e Configuração
INFINITE	NVENC-BUG	ALLOC	UNLOCK	SWAP
MONO		BCLEAN		NOBREAK
SLEEP		STATIC		LOGIC

Na categoria *Uso de CPU* estão os defeitos que alteram o uso do processador pela aplicação, seja ao realizar processamento desnecessário, como no mutante *Infinite*, ou ao não utilizar todo o hardware disponível, como no mutante *Mono*. A categoria *Uso de Memória* contém os defeitos que afetam a quantidade e forma de uso da memória principal do computador, como defeitos relacionados a vazamentos de memória, por exemplo. A categoria *Uso de GPU* abrange defeitos semelhantes às duas categorias anteriores, mas focando no uso do processamento e memória de vídeo, como erros de uso da plataforma de programação CUDA. As duas últimas categorias são a categoria *Sincronização*, relacionada a defeitos de uso e sincronização de *threads*, e a categoria *Parâmetros e Configuração*, com defeitos que alteram a execução e processamento da entrada pela aplicação.

5.3 Metodologia dos experimentos

Nesta seção será apresentado o ambiente de execução dos experimentos, descrevendo os componentes de hardware e software utilizados. Também serão apresentadas as etapas de execução dos experimentos e os parâmetros definidos para essas execuções, além de uma atenção especial à etapa final de agrupamento e análise, que é a responsável por gerar os resultados da metodologia de testes *Tricorder*.

5.3.1 Ambiente de execução

Os experimentos foram realizados utilizando um computador pessoal e as etapas de agrupamento e análise foram auxiliadas pelo uso de computadores disponibilizados no LaSDPC.

As etapas de monitoração da execução das aplicações testadas foram realizadas em um mesmo computador pessoal, devido ao período de *Home Office* durante o trabalho. Durante o período de execução dos experimentos, não foram realizadas modificações da especificação de hardware do computador. Esta máquina possui processador AMD *Quad Core* com *clock* base de 3.6*Ghz*, 16*GB* de memória RAM DDR4 com frequência de 2666*MHz* e uma placa gráfica Nvidia GTX 1660 com *clock* base de 1530*MHz* e 8*GB* de VRAM GDDR5.

Foram utilizadas duas unidades de armazenamento, sendo um Disco de Estado Sólido (SSD) SATA e um Disco Rígido (HD) SATA a 7200RPM. Nessas unidades de armazenamento estavam instalados os Sistemas Operacionais utilizados durante os experimentos, sendo o Sistema Operacional *Windows 10* instalado no SSD e o Sistema Operacional *Ubuntu 20.04* instalado no HD. O Primeiro e o Terceiro experimentos (Criptografia e OBS Studio) foram executados no Sistema Operacional *Ubuntu* e as aplicações instaladas no HD, e o Segundo e o Quarto experimentos (Ordenação e Simulação de Fluidos) foram executados no Sistema Operacional *Windows* e as aplicações instaladas no SSD. As unidades de armazenamento para a instalação dos sistemas operacionais foram escolhidas de acordo com a disponibilidade de espaço livre nas unidades e do uso pessoal do equipamento.

As etapas de Agrupamento e Análise foram realizadas em parte no computador pessoal descrito acima e em um *cluster* de quatro computadores disponibilizados no LaSDPC (referenciados como *Cluster* a partir deste ponto), ambos com Sistema Operacional Ubuntu. O acesso ao *Cluster* foi realizado remotamente e seu uso foi necessário para o Terceiro e Quarto experimentos, devido ao aumento da complexidade das aplicações e de seus defeitos, além de vários testes realizados com a metodologia. Apesar de o tempo necessário para cada etapa do agrupamento ser maior nos computadores do *Cluster* em relação ao computador pessoal, a disponibilidade de quatro computadores executando diferentes etapas da análise aumentou a vazão dos agrupamentos, reduzindo o tempo total necessário para a execução dos mesmos.

Buscou-se manter um ambiente controlado para as execuções das aplicações em teste na etapa de monitoração. O computador onde foram realizadas as monitorações das aplicações ficou

dedicado exclusivamente para esta tarefa durante toda a execução do *Script* de monitoração, o que inclui todas as execuções das versões da aplicação indicadas. De forma a reduzir o impacto de aplicações em segundo plano e do Sistema operacional em Execução, o acesso à internet da máquina foi desabilitado durante as execuções e o computador reiniciado antes da execução do *Script*.

5.3.2 Execução dos experimentos

A monitoração realizada neste trabalho foi baseada nas etapas de monitoração dos experimentos realizadas no trabalho prévio, descritas no Capítulo 4, com as alterações necessárias para cada aplicação, considerando recursos específicos como a placa de vídeo e os testes que se fizeram necessários para este trabalho. Os *Scripts* de monitoração do uso de recursos utilizados durante este trabalho foram adaptados a partir dos *Scripts* em *Python* utilizados no trabalho prévio.

Os *Scripts* de monitoração permitem a personalização de parâmetros como a quantidade de execuções da aplicação testada, e a quantidade e intervalo de amostras de uso de recursos observados nas execuções.

As quantidades de execuções das diferentes versões testadas da aplicação impacta diretamente nas etapas de Agrupamento e Análise. Esse parâmetro indica a quantidade de arquivos utilizados como base dos agrupamentos, no caso das versões originais da aplicação testada, e a quantidade de arquivos de uso de recursos gerados para cada nova versão da aplicação, como os mutantes gerados no caso deste trabalho. O limite máximo de iterações da etapa de agrupamento é indicado pela quantidade de execuções monitoradas para as novas versões da aplicação, com a *Tricorder* não indicando a presença de defeitos caso todas essas execuções forem agrupadas nos mesmos grupos das execuções da versão original.

O intervalo entre a obtenção de amostras do uso de recursos indica para as bibliotecas de monitoração qual o tempo em milissegundos entre cada nova amostra gerada. A quantidade de amostras obtidas indica, em número de linhas, o tamanho do arquivo de uso de recursos gerado. A combinação entre esses dois parâmetros indica o tempo, em milissegundos, de execução da aplicação enquanto é realizada a amostragem de uso de recursos, com o cálculo desse tempo sendo realizado pela multiplicação do intervalo entre amostras e a quantidade de amostras.

Durante a execução do primeiro experimento (Criptografia) percebeu-se uma alteração do uso de recursos das primeiras execuções da aplicação, principalmente na primeira execução, onde a aplicação está sendo carregada na memória principal pela primeira vez além de possíveis operações de *cache* executadas nesse momento. Para contornar este problema, foram inseridas execuções de *warmup* (aquecimento) antes de iniciar a monitoração da aplicação. Assim, foram realizadas três execuções de *warmup*, com a possibilidade de personalização dessa quantidade, com cada uma executando pelo mesmo tempo das execuções normais porém sem realizar

a monitoração de recursos. O objetivo das execuções de *warmup* é evitar que os processos realizados durante as primeiras execuções não afetem os arquivos de uso de recursos utilizados nas etapas de Agrupamento e Análise, diminuindo a possibilidade dessas alterações resultar em um falso positivo ou falso negativo para a presença de defeitos.

O uso da biblioteca *psutil* de monitoração de recursos foi mantido, com as mesmas métricas utilizadas no trabalho prévio, adicionando-se a métrica quantidade de Bytes transferidos. Com a adição desta métrica, a biblioteca *psutil* foi a responsável pela monitoração de quatro métricas de uso de recurso, sendo elas a porcentagem de uso do processador (CPU), a quantidade de memória principal utilizada em Bytes (RAM), a quantidade de operações de entrada e saída (IO) e a quantidade de dados transferidos pela entrada e saída em Bytes (IOBytes).

A biblioteca *psutil* não realiza a monitoração de métricas relacionadas ao uso de GPU, então foi necessária a busca por uma ferramenta auxiliar para esta demanda. Primeiramente foi testado o uso da biblioteca *GPUtil*, que realiza a monitoração da porcentagem de uso do processador gráfico e uso de memória de vídeo em bytes. Foi observado um problema na sincronização das taxas de amostragem da biblioteca *psutil*, que permite o usuário selecionar o intervalo entre as amostras enquanto a biblioteca *GPUtil* não permite essa configuração e afeta a taxa de amostragem da biblioteca *psutil*. Buscando resolver este problema, foi escolhida a biblioteca *nvidia_smi* (NVIDIA, 2022b) para a monitoração das métricas de porcentagem de uso do processador gráfico e uso de memória de vídeo em bytes, que não afeta a taxa de amostragem da biblioteca *psutil*.

Seguindo a mesma metodologia de funcionamento dos *Scripts* de monitoração do trabalho prévio, os dados de uso de recursos são gravados em um arquivo de valores separados por vírgula (CSV) ao final de cada execução da aplicação em teste. Cada linha do arquivo contém uma amostra obtida pelas bibliotecas de monitoração e as métricas observadas, sejam quatro ou seis métricas dependendo da aplicação, são salvas em uma coluna deste arquivo.

Durante o desenvolvimento do trabalho observou-se que os valores das métricas são salvos como números com diferentes tamanhos, como um número longo de quantidade de Bytes da memória RAM ou um número mais curto da porcentagem de uso da CPU, por exemplo. Levantou-se a hipótese de que essa diferença de tamanhos da representação das métricas influencia na priorização de uma métrica sobre as demais durante a etapa de agrupamento, uma vez que os métodos de agrupamento analisam os arquivos como um todo, sem observar as particularidades de cada uma das métricas e, portanto, dando mais prioridade para a alteração de valores representados por números mais longos. Para contornar este possível problema, realizaram-se testes de como realizar alguma normalização das representações das métricas de uso de recursos. A primeira opção escolhida foi a de modificar as unidades de representação das métricas mais longas, especialmente a quantidade de memória RAM e memória de vídeo utilizadas, passando da representação de Bytes para a representação de *KiloBytes* e posteriormente de *MegaBytes*.

A partir do resultado obtido dessa primeira modificação na representação das métricas,

testou-se o uso da representação da taxa das métricas em relação ao maior valor observado para a métrica durante a execução da aplicação sem defeitos. Os novos arquivos de uso de recursos gerados por este método têm como base os arquivos de uso de recursos gerados pelo método anterior, com métricas de tamanhos semelhantes. Os arquivos de uso de recursos das execuções da aplicação original são analisados em busca do maior valor obtido para cada métrica, que são utilizados em um próximo passo onde todas as amostras passam a ser representadas como uma razão entre o valor da amostra e o maior valor da métrica obtido no passo anterior. Esse método produz novos arquivos de uso de recursos onde todos os valores são representados com a mesma quantidade de dígitos e, geralmente, estão contidos entre 0 e 1 ou valores pouco maiores que 1. Foi realizado um teste com o uso de dados de monitoração normalizados, com uma quantidade menor de execuções e o uso desse método apresentou afetou os resultados de forma negativa, como é possível observar nas Tabelas 8, 9, 10 e 11.

Tabela 8 – Experimento com 20 execuções da base e dos mutantes - Representação inteira

	Mutante			
Entrada	**BCLEAN-GIT**	**BCLEAN-MOD**	**SLEEP-START**	**SLEEP-LOOP**
NVENCL	N	8	3	3
NVENCM	13	8	9	9
NVENCH	6	10	8	8
SOFTWL	N	N	N	N
SOFTWM	6	N	7	7
SOFTWH	N	17	N	N

Tabela 9 – Experimento com 20 execuções da base e dos mutantes - Representação inteira

	Mutante		
Entrada	**NOBREAK**	**NVENC-BUG**	**UNLOCK**
NVENCL	3	13	6
NVENCM	13	6	7
NVENCH	8	8	8
SOFTWL	N	-	5
SOFTWM	7	5	8
SOFTWH	N	-	2

Tabela 10 – Experimento com 20 execuções da base e dos mutantes - Representação normalizada

	Mutante			
Entrada	**BCLEAN-GIT**	**BCLEAN-MOD**	**SLEEP-START**	**SLEEP-LOOP**
NVENCL	18	N	N	N
NVENCM	N	N	16	N
NVENCH	16	8	9	10
SOFTWL	N	17	N	19
SOFTWM	8	N	10	N
SOFTWH	N	19	11	12

Tabela 11 – Experimento com 20 execuções da base e dos mutantes - Representação normalizada

	Mutante		
Entrada	**NOBREAK**	**NVENC-BUG**	**UNLOCK**
NVENCL	N	6	17
NVENCM	17	N	18
NVENCH	10	8	N
SOFTWL	N	-	N
SOFTWM	11	16	12
SOFTWH	N	-	16

A partir desses dois testes decidiu-se por utilizar o primeiro método descrito acima, com as métricas representadas por números de tamanhos semelhantes. Este método foi o que apresentou um melhor resultado de uma forma geral tanto para a detecção da presença dos defeitos ativados quanto para a não detecção quando não há um defeito sendo executado.

5.3.3 Parâmetros de execução definidos

A quantidade de amostras observadas, o intervalo entre cada amostra e, por consequência, o tempo de execução da aplicação em teste foram mantidos com os mesmos valores utilizados no trabalho prévio, com 300 amostras observadas em 30 segundos de execução a uma taxa de uma amostra a cada 100 milissegundos.

Foram realizados testes no terceiro experimento (OBS Studio) alterando o número de execuções da versão original e o número de execuções das versões mutantes em determinados casos. A configuração utilizada no trabalho prévio tinha como parâmetro 30 execuções das versões original e mutantes.

Primeiramente foram realizados testes monitorando um número maior de execuções da versão original da aplicação, utilizando 50, 75, 100 e 125 execuções, com o objetivo de aumentar a quantidade de dados da base de aprendizado para o método de Agrupamento e, assim, diminuir a taxa de erros da *Tricorder*. Esses testes foram realizados para o mutante Controle, uma versão sem defeitos, com poucos benefícios de resultado ao aumentar essa quantidade mas com um aumento considerável no tempo de execução das etapas de agrupamento e análise, piorando assim a relação custo x benefício neste caso.

Para a geração dos perfis de uso de recursos foram monitoradas quatro ou seis métricas de uso de recurso, dependendo da necessidade da aplicação testada. Todos os experimentos realizados monitoraram a porcentagem de uso do processador, a quantidade em bytes de memória principal utilizada, a quantidade de operações de entrada e saída e a quantidade em bytes de entrada e saída transferidos. O terceiro experimento foi o único onde foram monitoradas seis métricas, já que a aplicação OBS Studio utiliza a placa de vídeo durante sua execução, monitorando também as métricas de porcentagem de uso do processador gráfico, a quantidade em bytes de memória de vídeo utilizada.

5.3.4 Agrupamento e Análise

A etapa de agrupamento e análise é onde acontece o processamento dos dados gerados pela etapa de monitoração, gerando os resultados sobre a presença de defeitos. Os agrupamentos são gerados com a ferramenta *Damicore* a partir da execução do *Script* de análise, que gerencia os parâmetros do agrupamento e a escolha dos arquivos que serão agrupados.

O *Script* de análise realiza os agrupamentos separadamente para cada combinação de mutante e carga de trabalho, tendo como base todos arquivos de uso de recursos gerados pela monitoração da versão original da aplicação em teste. São gerados os grupos com as execuções da versão original e são adicionados um a um os arquivos de monitoração do mutante analisado, executando a ferramenta de agrupamento a cada iteração.

Os agrupamentos gerados com um novo arquivo de monitoração do mutante são analisados buscando a presença de um grupo formado apenas por execuções do mutante. Um grupo com essa característica indica que essas as execuções da versão mutante com a carga de trabalho utilizada, apresentam um perfil de uso de recursos diferente do perfil das execuções da aplicação original com todas as cargas de trabalho monitoradas.

São utilizadas duas heurísticas como critério de parada, as mesmas utilizadas no trabalho prévio. A **Heurística Simples** interrompe a etapa de agrupamento na primeira iteração em que for encontrado um grupo formado apenas por execuções do mutante, enquanto a **Heurística Restritiva** necessita de duas iterações seguidas com a presença desse tipo de grupo. Durante a realização dos experimentos foram observados casos onde a condição de parada da Heurística Simples foi atendida em uma certa iteração e a condição de paradas da Heurística Restritiva foi atendida apenas em iterações posteriores ou então não foi atendida até o fim da análise, com casos em que a primeira condição foi atendida mais de uma vez antes da segunda condição ou do fim das iterações.

O número de iterações de agrupamentos gerados é limitado pela quantidade de arquivos de monitoração gerados para a versão mutante: 30 iterações no caso dos experimentos realizados neste trabalho. Caso a análise não cumpra o critério de parada até o limite final das iterações as execuções da versão mutante da aplicação são consideradas livres de defeitos.

5.4 Aplicações testadas

Um programa que possua muitas alterações no seu comportamento ao executar uma mesma tarefa com uma determinada carga de trabalho pode dificultar o agrupamento realizado pela metodologia. Apesar do alto uso de recursos de uma aplicação facilitar o uso da metodologia *Tricorder*, programas com baixo uso de recursos também podem ser testado com a *Tricorder*. Este é o caso do OBS Studio, por exemplo, que é utilizado para a gravação ou transmissão da tela de um computador executando uma outra aplicação mais suscetível à variações de desempenho

como jogos, por exemplo.

Como citado anteriormente, aplicações com uso consistente dos recursos do sistema são o principal alvo da metodologia. De forma complementar a essa consistência, também é esperado que uma aplicação testada pela *Tricorder* possua uma escalabilidade do uso dos recursos, de cargas de trabalho mais leves a cargas mais pesadas ou com diferentes porções de código executadas, apresentando diferentes perfis de uso de recurso para diferentes cargas de trabalho. A etapa de agrupamento não reúne as execuções de uma mesma carga de trabalho necessariamente em um só grupo ou em grupos exclusivos, porém isso não afeta a execução da metodologia.

Aplicações com execuções periódicas, ou seja, aplicações que executam um mesmo trecho de código várias vezes, permitem que um possível defeito na região repetida do código execute mais vezes durante sua execução. Um defeito que afete o uso de recursos da aplicação que seja ativado diversas vezes durante o período de monitoração, pode facilitar a possibilidade de sua detecção pelo teste com a metodologia *Tricorder*.

O objetivo da *Tricorder* é revelar defeitos em programas de computador. Para tanto, os experimentos de teste da metodologia necessitam da presença de um defeito conhecido em uma versão da aplicação, além de uma versão da aplicação considerada como livre de defeitos. Para inserir os defeitos nas aplicações, precisou-se ter acesso ao código fonte da aplicação para modificá-lo, sendo as aplicações de código aberto utilizadas neste trabalho. O Github (GITHUB, 2022) foi a plataforma utilizada para encontrar essas aplicações, pela facilidade de uso, além da grande quantidade e variedade de aplicações de código aberto disponibilizadas. O uso dessa plataforma também permite encontrar aplicações relativamente conhecidas, com um grande número de usuários e muitos desenvolvedores contribuindo em todo o seu ciclo de vida.

Um fator determinante para a escolha das aplicações utilizadas é a sua facilidade de instalação, compilação e uso. Aplicações com instalação e compilação muito complexas dificultam a sua utilização, além do tempo necessário para realizar uma execução e exploração antes de sua escolha para o teste.

A escolha das aplicações foi realizada considerando os fatores abaixo. Os primeiros fatores levados em consideração foram o uso consistente e escalável de recursos, execuções periódicas e de código aberto. Além destes, a relevância dos projetos na plataforma GitHub também influenciou na escolha das aplicações, priorizando aplicações com melhor classificação pelos usuários da plataforma.

5.4.1 Criptografia

O primeiro nicho de aplicações utilizado nos experimentos deste trabalho foram as aplicações de criptografia de dados. Estas aplicações têm a característica de alto uso de recursos do computador, principalmente o uso do processador, assim como a possibilidade de processar

vários tipos de entradas e processar algumas dessas entradas em sequência. Essas características de aplicações de criptografia de arquivos cumprem as demandas de alto uso de recursos, escalabilidade e execuções periódicas, indicadas na primeira parte dessa seção, com a aplicação escolhida devendo possuir código aberto e uma usabilidade relativamente simples para o uso no experimento.

A aplicação Crypto-Example (TULLY, 2013) foi escolhida para a realização do primeiro experimento. Essa aplicação foi implementada como uma prova de conceito dos métodos de criptografia *RSA* e *AES*, usando como base a biblioteca *OpenSSL* (OPENSSL, 2022). O autor da aplicação relata que teve uma experiência prévia com uma implementação do método *RSA* utilizando o *OpenSSL* e implementou esta aplicação com o objetivo de utilizar posteriormente em outro projeto.

5.4.2 Java - Ordenação

Para o segundo experimento procurou-se testar o uso da *Tricorder* com uma linguagem de programação diferente das linguagens C e C++ já utilizadas. A linguagem Java foi a escolhida para a realização do experimento, uma vez que apresenta características diferentes das linguagens já utilizadas, como ser executada na Máquina Virtual Java (*JVM*) e possuir coletor de lixo, por exemplo. Buscou-se por uma aplicação que fosse simples, porém que apresentasse as características desejadas, de forma a isolar a linguagem de programação como variável mais importante do experimento.

As implementações Java dos métodos de ordenação *MergeSort* e *QuickSort* foram escolhidas em suas versões paralelas, apresentadas por Shaowdy (2017). Essa aplicação possui alto uso de recursos e escalabilidade desse uso, de acordo com a carga de trabalho utilizada. Além disso, possui mais de um método de ordenação, o que possibilita um novo ponto de variação de cargas de trabalho.

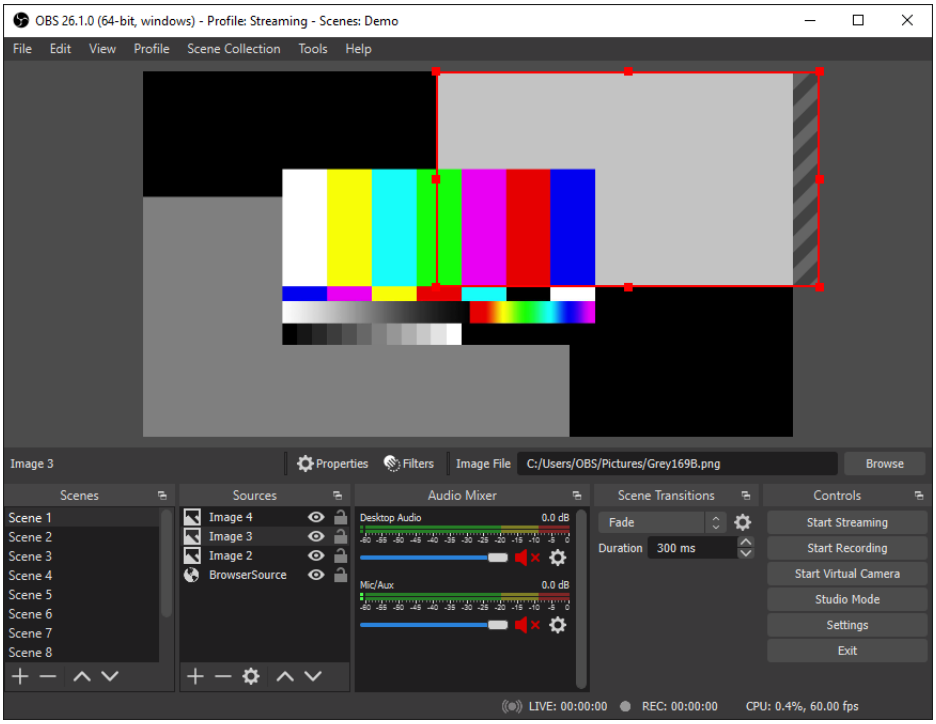
5.4.3 Gravação de vídeos

No terceiro experimento optou-se por realizar o teste de uma aplicação mais complexa. A partir de pesquisas realizadas no Github, foram encontradas aplicações maiores, conhecidas e relevantes em seus nichos. Estas serão citadas nas próximas seções. Dentre as aplicações encontradas escolheu-se a aplicação OBS Studio (OBS, 2022), um software livre projetado para capturar, compor, codificar, gravar e transmitir conteúdo de vídeo de forma eficiente.

O OBS Studio é uma das ferramentas de captura e transmissão de vídeos mais populares atualmente, usado tanto por criadores de vídeos e transmissões ao vivo para a internet quanto por educadores para a elaboração de vídeo-aulas e materiais de ensino. A Figura 9 ilustra a interface gráfica do OBS Studio. Esta ferramenta visa apresentar alto desempenho de captura e mixagem de vídeo e áudio em tempo real, possibilitando a criação de cenas a partir de múltiplas fontes,

como captura de janelas, câmeras e arquivos (OBS, 2022).

Figura 9 – Demonstração da interface gráfica do OBS Studio.



Fonte: OBS (2022).

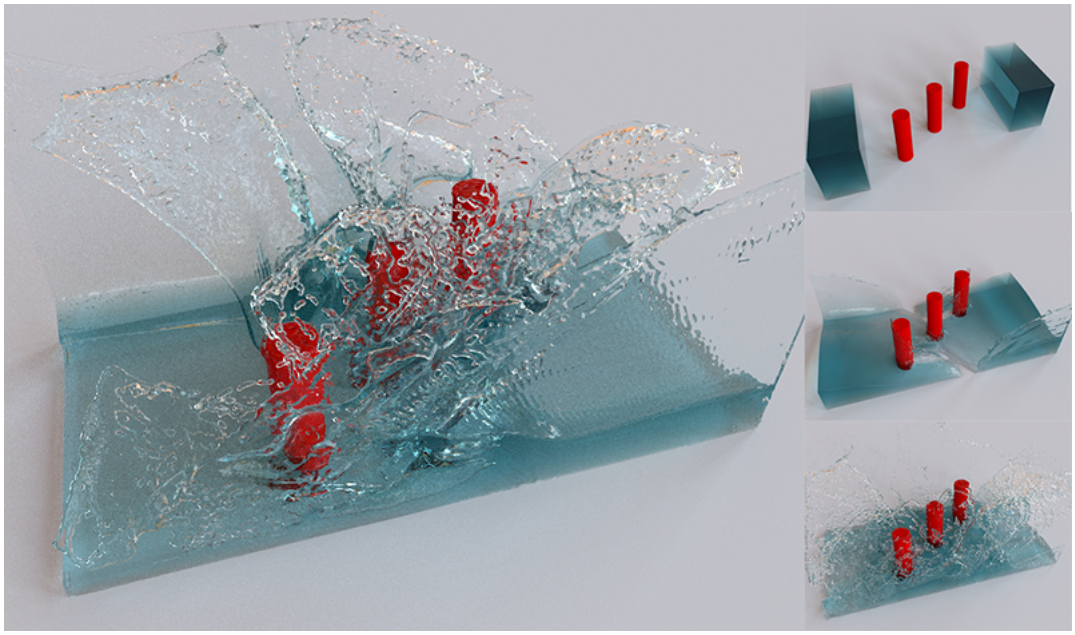
Dentre seus recursos, o OBS Studio fornece a possibilidade de criação de cenas com a possibilidade de alternância com transições personalizadas. Também é possível destacar a interface intuitiva, que permite ao usuário configurar parâmetros de áudio, vídeo e uso de hardware sem muito esforço. Além do usuário básico, também são oferecidas configurações mais avançadas, como configurações avançadas de vídeo, perfis de uso da aplicação ou parâmetros de linha de comando.

5.4.4 Simulação de fluidos

O quarto experimento escolhido foi uma aplicação real, encontrada em pesquisas no Github, e com as características citadas na seção anterior. Dentre os nichos de aplicação pesquisados, foram pesquisados diferentes tipos de simulações computacionais. Dentre estas aplicações, optou-se por um motor de simulação de fluidos por partículas e malhas descrito em (KIM, 2017). Segundo o autor, a proposta aborda a dinâmica de fluidos de forma orientada ao código, explorando técnicas de simulação e discutindo questões práticas de design e otimizações (KIM, 2017). As implementações estão disponibilizadas em um repositório no Github, com atualizações recentes e possibilidade de discussões sobre o tema.

A aplicação *Jet framework* (KIM, 2022) é um motor de simulação de fluido para aplicações gráficas computacionais. São apresentadas implementações de diversos métodos de

Figura 10 – Exemplo de simulação com o método PIC



Fonte: Kim (2017).

simulação de fluidos, como *SPH*, *PIC*, *FLIP* e *APIC*, assim como simulações de fumaça, resolvedores de sistemas lineares dentre outras ferramentas relacionadas. O repositório também fornece exemplos de uso dos métodos de simulação, com implementações bidimensionais e tridimensionais.

5.4.5 Nichos buscados mas não usados

As pesquisas por aplicações para testes nos experimentos resultaram em um bom número de aplicações interessantes, relevantes e com algum potencial de uso da metodologia *Tricorder*. Essas aplicações não foram utilizadas neste trabalho em função do tempo.

Um nicho de aplicações com alto uso de recursos são os *benchmarks* de hardware. Foram realizados alguns testes preliminares com as aplicações *Cinebench R20* (MAXON, 2022), *Vulkan API* (GROUP, 2022) e *Heaven Benchmark* (UNIGINE, 2022), mas estas possuem uma maior complexidade para a instalação e uso da API *Vulkan*, além de uma não trivial definição de cargas de trabalho para essas aplicações. O objetivo dessas aplicações é realizar um teste do hardware utilizado, aplicando uma carga de trabalho alta e constante, com poucas possibilidades de adaptações para o uso da metodologia *Tricorder*.

Outros nichos foram brevemente estudados, como aplicações de *Blockchain*, edição de imagens, jogos e sistemas gerenciadores de bancos de dados. Algumas dessas aplicações foram selecionadas como um eventual "*backup*", caso alguma das aplicações previamente escolhidas apresentasse problemas ou caso houvesse tempo hábil para um novo experimento. Dentre essas aplicações estão o *Simcoin* (SBA, 2022), o *MozJPEG* (MOZILLA, 2022), os jogos *Mindustry*

(ANUKEN, 2022) e *Godot* (LINIETSKY, 2022), e os SGBDs *Postgres* (POSTGRESQL, 2022) e *MongoDB* (MONGODB, 2022).

5.5 Defeitos inseridos nas aplicações testadas

Esta seção apresenta as versões das aplicações utilizadas nos experimentos com e sem defeitos inseridos. Em função do uso do Teste de Mutação como inspiração para a criação das versões modificadas (com defeitos) das aplicações testadas, essas versões são chamadas neste trabalho como *Mutantes*. Os Mutantes estão separados nas categorias *Aplicação Original*, *Controle* e *Mutantes Defeituosos*, com representantes dessas categorias presentes em todos os experimentos realizados com o mesmo objetivo de uso.

A aplicação original é uma versão sem defeitos conhecidos. Esta versão original foi usada tanto no treinamento inicial, quando nas execuções do Grupo Controle, onde é esperado que essas duas versões tenham perfis de uso de recursos semelhantes por executarem um mesmo código, apesar de existir uma pequena variação entre cada execução de um mesmo código. Para as versões com defeitos inseridos é esperado que tenham o perfil de uso de recursos alterado em relação à aplicação original, dependendo do defeito implementado e da carga de trabalho utilizada. Ressalta-se que cada versão mutante das aplicações possui até um defeito conhecido e inserido propositalmente no código, como descrito nas seções a seguir.

5.5.1 Aplicação original

A primeira versão utilizada de uma aplicação em teste pela metodologia *Tricorder* é a sua versão Original, considerada "sem defeitos". Essa é a versão que terá suas execuções monitoradas para formar a base inicial dos agrupamentos formados para a etapa de análise.

Nos experimentos realizados, essa versão da aplicação é utilizada para os primeiros testes de seu funcionamento Com a aplicação Original e a ferramenta de monitoração configuradas, são realizados testes sobre o uso de recurso da execução da aplicação com diferentes cargas de trabalho de forma a caracterizar o seu comportamento e identificar o impacto de cada mudança de carga de trabalho.

A partir da caracterização do uso de recursos pela aplicação, são definidas as cargas de trabalho que serão utilizadas no experimento. Neste trabalho procurou-se continuar o padrão de cargas de trabalho seguido pelo trabalho prévio (MONTES, 2019a), com dois tipos de cargas diferentes e três níveis de complexidade em cada um.

5.5.2 Controle

Assim como a versão Original, a versão Controle também é considerada sem defeitos e é simplesmente uma cópia da aplicação Original. Nos experimentos, por simplicidade na descrição,

essa versão é considerada é a primeira versão mutante do experimento, apesar de não apresentar defeitos conhecidos *Tricorder*.

A versão de controle visa verificar se a aplicação mantém o seu perfil de uso de recursos ao ser executada diversas vezes. Essa verificação é feita ao comparar as execuções da aplicação entre si, com agrupamentos formados com as versões Original e Controle da aplicação.

São esperados resultados negativos para a presença de defeitos pela metologia *Tricorder* para o mutante Controle, uma vez que não foram inseridos defeitos e o comportamento da aplicação não deve se alterar significativamente nessas execuções. Podem ocorrer falsos positivos nesta etapa em algum caso em que uma ou mais execuções da versão Controle sofra uma alteração para que a ferramenta de agrupamento acuse que foi o suficiente para a formação de um novo grupo isolado.

Um número alto de falsos positivos pode indicar que alguma configuração da metodologia ou do ambiente de execução não estão adequados e devem ser ajustados, ou que a aplicação sofre alterações no uso de recursos entre execuções normais e, assim, a metodologia *Tricorder* pode não ser adequada para o seu teste. Durante a execução dos experimentos descritos neste trabalho foram encontradas situações que se encaixam na primeira opção, com ajustes de configurações realizados; mas não foram encontradas situações do segundo tipo (nestes experimentos), com aplicações inadequadas para o uso da metodologia.

5.5.3 Sleep

O mutante *Sleep* foi o primeiro mutante utilizado em todos os experimentos realizados. O defeito causado por este mutante é a execução de um comando *sleep*, com sua devida implementação de acordo com a linguagem de programação utilizada. O comando *sleep* deixa a aplicação ou *thread* com execução parada, reduzindo o consumo de recursos, por um determinado período de tempo antes de continuar sua execução normal.

A implementação do *Sleep* é relativamente simples, sendo que todas as linguagens utilizadas neste experimento possuem uma função ou método próprio para o *sleep*, com funcionamento direto e intuitivo. Ademais, também permitem a sua inserção em diferentes posições no código fonte e a possibilidade de alteração do tempo de efeito desejado.

5.5.4 Mutantes conhecidos

Após os primeiros testes da metodologia com a versão Controle e *Sleep*, são inseridos novos defeitos na aplicação (cada versão mutante possui, no máximo, um único defeito). Nesta fase foram inseridos alguns dos defeitos já utilizados em experimentos anteriores, como *Bclean*, *Infinite* ou *Mono*, por exemplo (MONTES, 2019a).

Com a experiência dos experimentos anteriores investigou-se o código fonte da aplicação, em busca de regiões do código com características propícias para a inserção desses defeitos,

como uma limpeza de memória para a inserção do *Bclean*, uma estrutura condicional (*Switch - case*) para a inserção do *Nobreak* ou uma região paralela onde o mutante *Mono* pudesse ser inserido sem impactos além dos previstos.

Os mutantes dessa categoria são os mesmos apresentados no Capítulo 4, e são implementados com as devidas adaptações necessárias para o seu funcionamento nas aplicações testadas (MONTES, 2019a). Esses defeitos possuem um comportamento conhecido e já testado com a metodologia *Tricorder*, além de serem aplicáveis em diversos nichos de aplicação e podem ser adaptados para outras linguagens de programação.

5.5.5 Defeitos reais da aplicação

Além dos defeitos citados no item anterior, procurou-se por defeitos reais da aplicação. Foram realizadas pesquisas nos repositórios do código fonte, buscando por algum defeito conhecido e corrigido da aplicação. Em um primeiro momento foram procurados defeitos que pudessem ser classificados em um dos mutantes conhecidos, como um vazamento de memória (*memory leak*) que tem o mesmo efeito do mutante *Bclean*, por exemplo.

Essa é a categoria de defeito de maior relevância para o teste da aplicação, uma vez que são defeitos reais da aplicação que foram encontrados com o uso de outros métodos de teste.

Nos experimentos realizados foi possível utilizar três defeitos reais da aplicação OBS Studio para implementar quatro versões mutantes, com bons resultados para a metodologia *Tricorder* que serão apresentados no Capítulo 8.

5.5.6 Defeitos novos

Além dos defeitos utilizados nos experimentos anteriores e dos defeitos reais encontrados nos repositórios das aplicações, também foram sugeridos alguns novos defeitos, inspirados nas taxonomias de defeitos estudadas previamente e também nos defeitos já utilizados antes na *Tricorder*, com alguma diferença no seu funcionamento mas comparáveis entre si. Também foram definidos defeitos completamente novos para o trabalho, tendo como base os nichos das aplicações ou a suas linguagens de programação, implementados a partir da observação do código fonte e de experiências anteriores dos integrantes do grupo de pesquisa.

Com a utilização dos defeitos já conhecidos, defeitos reais e os novos defeitos, os experimentos realizados neste trabalho abrangem todas as categorias de defeitos da classificação gerada a partir de taxonomias da literatura, respeitando as características das aplicações para a utilização dos defeitos.

CAPÍTULO

6

EXPERIMENTO CRIPTOGRAFIA

6.1 Considerações iniciais

O experimento com a aplicação Criptografia visa principalmente utilizar a metodologia *Tricorder* para um primeiro teste em outra aplicação, diferente do processamento de streams, realizado previamente em (MONTES, 2019a). Decidiu-se por escolher uma aplicação implementada nas linguagens C ou C++ para manter as linguagens utilizadas nos experimentos do trabalho prévio (MONTES, 2019a), evitando a escolha de uma linguagem que afetasse de alguma forma o funcionamento da metodologia. Foi dada preferência por um nicho de aplicação com consumo elevado de recursos, uso relativamente simples e com alguma facilidade de escolher cargas de trabalho de diferentes complexidades.

6.2 Aplicação

A partir das pesquisas por algoritmos de criptografia de arquivos no Github, foi escolhida a aplicação *Crypto-Example* (TULLY, 2013) para o primeiro experimento de teste da metodologia *Tricorder*. Esta aplicação utiliza a biblioteca *OpenSSL* (OPENSSL, 2022) para implementar um algoritmo de criptografia baseado no padrão *AES* (DAEMEN; RIJMEN, 2001), com métodos de compressão e descompressão de um arquivo de entrada. Como citado, a aplicação tem projeto público, com instruções de instalação e uso, além de um espaço para discussões.

Implementada em *C++*, a aplicação recebe o caminho de um arquivo como parâmetro de linha de comando e executa os métodos de encriptação e desencriptação do arquivo de entrada, salvando os dois arquivos de saída no diretório de trabalho. Por sua simplicidade, a aplicação possui uma limitação quanto ao tamanho do arquivo de entrada, sugerindo o uso de arquivos não muito grandes ou, se necessário, a divisão do arquivo em partes menores.

De forma a manter o padrão de utilizar nos experimentos com a *Tricorder* aplicações

com periodicidade de execução, foram realizadas modificações para a aplicação executar tendo um diretório como entrada e executar os métodos de encriptação e desencriptação para todos os arquivos desse diretório. Com tais modificações, a aplicação é executada em *loop* processando os arquivos do diretório, executando os dois métodos antes de passar para o próximo arquivo. A execução termina após o processamento de todos os arquivos de entrada.

6.3 Experimento

Para este experimento foram observadas quatro métricas de uso de recursos, as três métricas utilizadas no trabalho prévio (MONTES, 2019a), percentual de uso de CPU, uso de RAM (em MBytes) e quantidade de operações de Entrada e Saída, e a métrica de uso de Entrada e Saída (I/O) em Bytes. A monitoração do uso de recursos foi realizada utilizando um *script* em Python baseado no *script* utilizado no trabalho prévio, com adaptações para a execução de uma nova aplicação e para a observação de uma nova métrica de uso de recursos.

A etapa de monitoração seguiu o trabalho prévio como padrão (MONTES, 2019a). Cada execução da aplicação foi monitorada por 30 segundos, onde foram realizadas 300 amostras com intervalo de 100 milissegundos. A quantidade de execuções da versão original da aplicação sofreu uma modificação de 30 para 50 execuções, com o objetivo de melhorar a etapa de agrupamento ao disponibilizar uma base maior para o método de agrupamento. As versões mutantes, incluindo o mutante de controle, continuaram com 30 execuções monitoradas como padrão. A escolha dessas quantidades de execuções foi baseada em resultados preliminares, analisando-se o impacto dessas quantidades na eficácia do agrupamento.

As entradas da aplicação foram geradas a partir de um gerador de números aleatórios simples, implementado com a linguagem *C++*. O gerador de entradas cria vários arquivos de texto com os números gerados, variando a quantidade de números em cada arquivo e gerando uma quantidade de arquivos suficiente para a aplicação ser executada por um tempo maior do que os 30 segundos de monitoração do experimento.

Como observado na Tabela 12, são cinco cargas de trabalho que variam a quantidade de números aleatórios inseridos e, consequentemente, seu tamanho em disco. As cargas vão desde *Lowest*, com 15 milhões de números aleatórios que ocupam cerca de 55 Megabytes, até a carga *Highest*, com 35 milhões de números aleatórios que ocupam cerca de 129 Megabytes. Os limites máximo e mínimo e o intervalo entre as cargas foi definido a partir de testes com uma quantidade maior de entradas e de forma a balancear o tempo de execução gasto para criptografar e descriptografar um arquivo comum aumento gradual e perceptível do uso de recursos pela aplicação, com testes iniciais de cargas de trabalho variando entre 10 e 175 milhões de números aleatórios por arquivo.

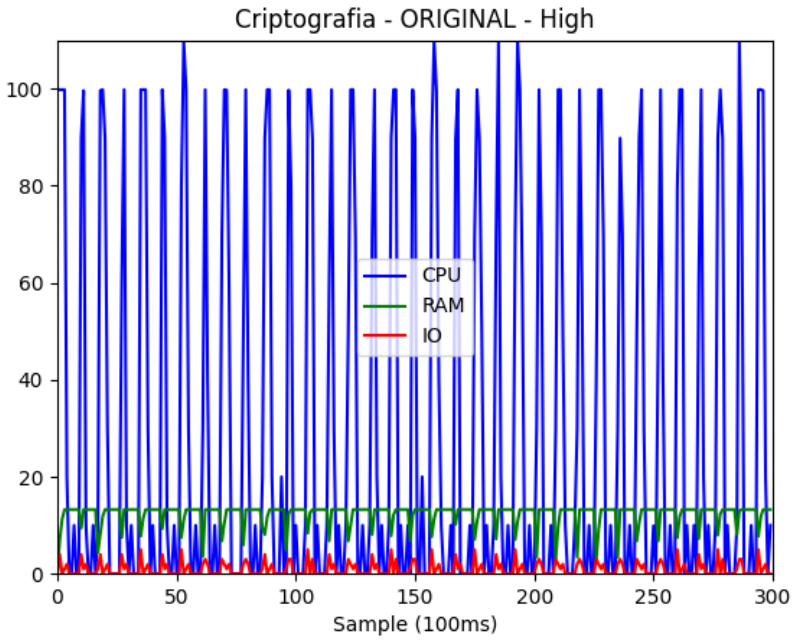
Tabela 12 – Cargas de trabalho definidas para a aplicação de criptografia

Carga de Trabalho	Quantidade de números	Tamanho
Lowest	15 Million	55 MB
Low	20 Million	74 MB
Medium	25 Million	93 MB
High	30 Million	111 MB
Highest	35 Million	129 MB

6.4 Mutantes

Para o teste da metodologia *Tricorder* com a aplicação de criptografia, foram implementadas quatro versões mutantes defeituosas. Utilizaram-se defeitos já conhecidos, baseados no trabalho prévio (MONTES, 2019a), e um novo defeito foi implementado a partir do estudo do código fonte da aplicação. Além das versões defeituosas, também foi utilizado um mutante de controle, sem a inserção de defeitos. A Figura 11 mostra o gráfico de uso de recursos da aplicação de criptografia e do mutante de controle.

Figura 11 – Gráfico de uso de recursos da aplicação original

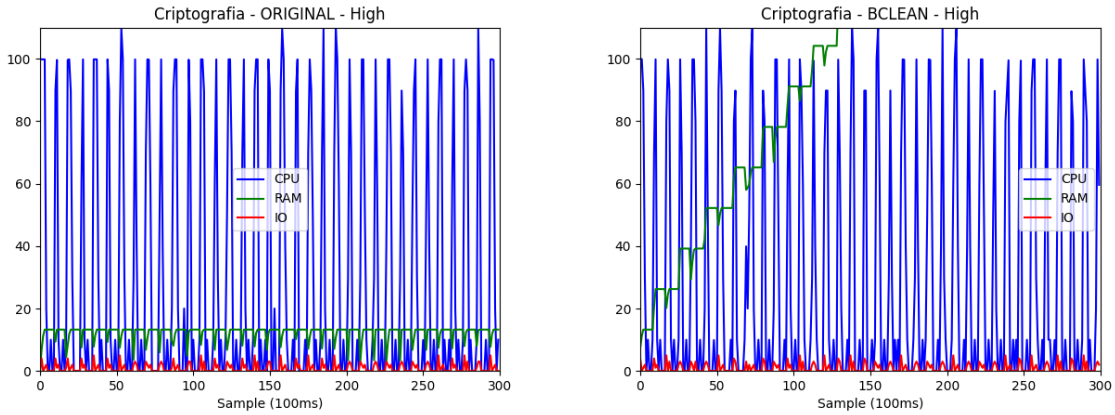


Fonte: Elaborada pelo autor.

A primeira versão mutante foi definida a partir de um defeito encontrado na aplicação, utilizando-se, de fato, a metodologia *Tricorder*! Ao realizar os testes da etapa de monitoração, observou-se um aumento constante do uso de memória pela aplicação, principalmente entre cada iteração do processamento de um novo arquivo. Este comportamento é característico de um vazamento de memória, o defeito representado pelo mutante *Bclean* já utilizado no trabalho prévio. Com o conhecimento desse defeito na aplicação, realizamos uma correção para as versões

livres de defeitos e utilizamos a versão com vazamento de memória como um mutante para o experimento. A Figura 12 mostra o gráfico de uso de recursos do mutante *Bclean*, onde é possível perceber um uso de memória crescente durante a execução.

Figura 12 – Execuções da aplicação original e aplicação com mutante *BCLEAN*

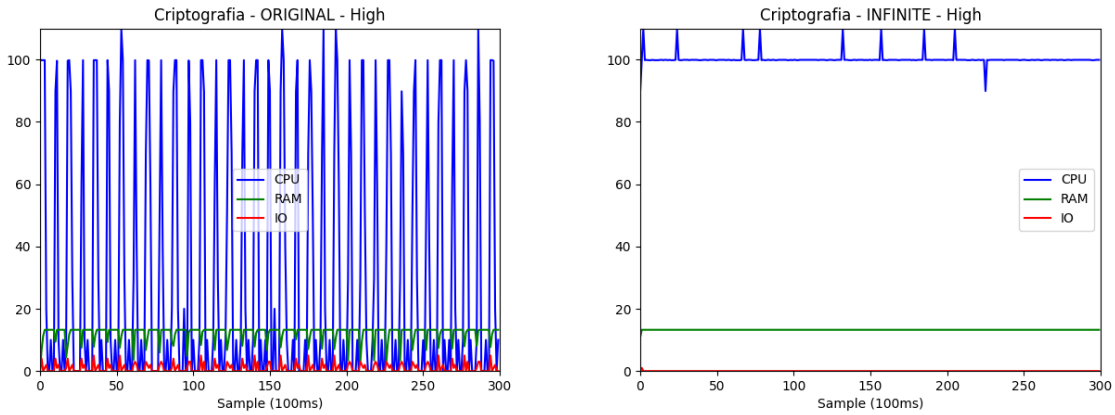


(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante BCLEAN

Fonte: Elaborada pelo autor.

Outros dois defeitos conhecidos foram utilizados para a implementação dos próximos mutantes. O primeiro desses defeitos foi representado pelo mutante *Infinite*, que simula um *loop* infinito em algum ponto da aplicação. Ao executar esse defeito, a aplicação entra em *loop* infinito, aumentando o uso de processador. A Figura 13 mostra o gráfico de uso de recursos do mutante *Infinite*, com um uso constante de toda a capacidade do processador.

Figura 13 – Execuções da aplicação original e aplicação com mutante *INFINITE*



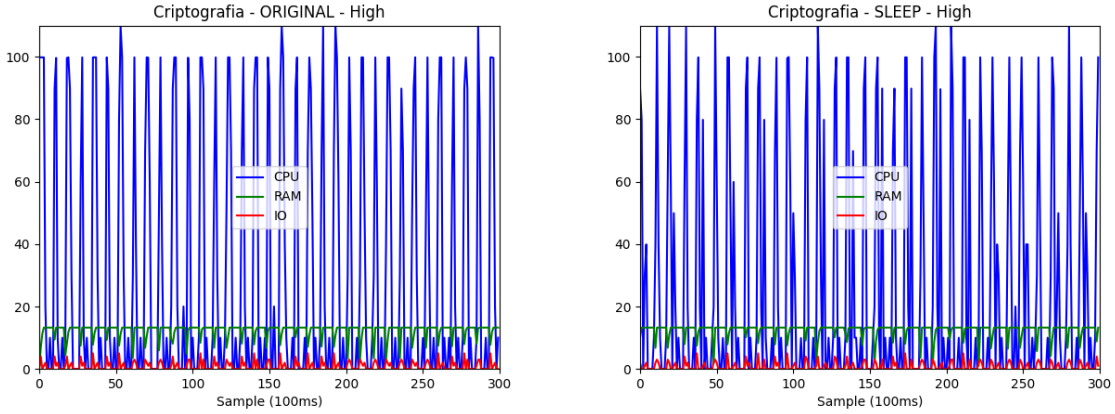
(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante INFINITE

Fonte: Elaborada pelo autor.

Outro defeito utilizado foi o *Sleep*. Ao executar o defeito o *Sleep*, a aplicação tem sua execução suspensa por 100 milissegundos, afetando o uso de recursos do computador,

principalmente o uso do processador. A Figura 14 mostra o gráfico de uso de recursos do mutante *Sleep*, ficando visíveis as alterações no uso do processador com novos períodos de baixo processamento.

Figura 14 – Execuções da aplicação original e aplicação com mutante *SLEEP*

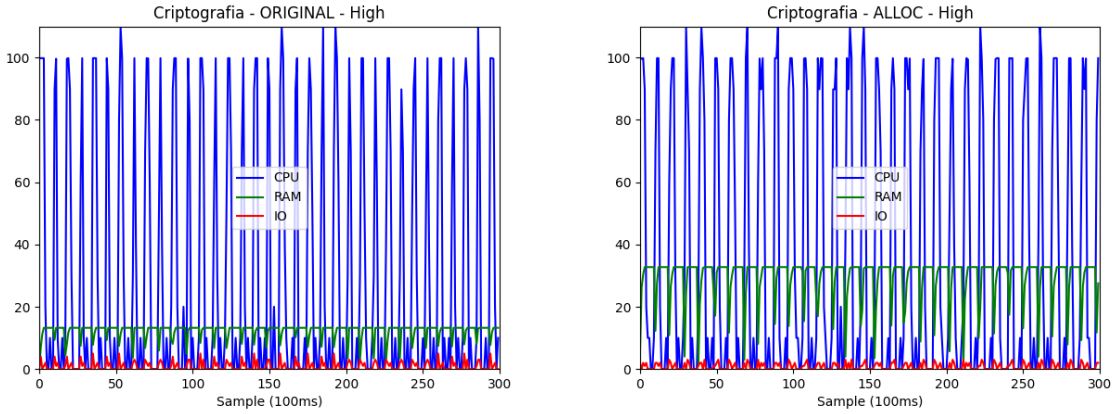


(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante SLEEP

Fonte: Elaborada pelo autor.

O último mutante (*Alloc*) considerou um defeito novo, a partir do estudo da aplicação utilizada e de discussões no grupo de pesquisa. O mutante *Alloc* possui uma alteração no trecho de código onde ocorre a alocação de memória para a mensagem criptografada, adicionando um comando para definir as posições de memória com zeros, o que modifica o comportamento da aplicação. A Figura 15 mostra o gráfico de uso de recursos deste mutante, onde há um uso maior de memória com um perfil semelhante à aplicação original.

Figura 15 – Execuções da aplicação original e aplicação com mutante *ALLOC*



(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante ALLOC

Fonte: Elaborada pelo autor.

6.5 Resultados

A Tabela 13 mostra os resultados do experimento Criptografia, utilizando a heurística simples para a detecção de anomalias. As linhas representam as cargas de trabalho utilizadas e as colunas representam os defeitos implementados. As células estão coloridas de acordo com o resultado obtido: células em vermelho indicam que a metodologia não acusou a presença do defeito inserido e as células em amarelo representam acertos da metodologia, onde o número na célula indica em qual execução a metodologia acusou a presença do defeito (quanto menor o número na detecção de defeitos, melhor).

Tabela 13 – Resultados do experimento da aplicação de criptografia - Heurística Simples

	Mutante			
Entrada	**BCLEAN**	**INFINITE**	**SLEEP**	**ALLOC**
Lowest	4	5	N	4
Low	3	7	30	3
Medium	3	4	14	4
High	4	4	N	3
Highest	3	3	25	4

Com os resultados apresentados, é possível observar um bom desempenho da metodologia com os mutantes *Bclean*, *Infinite* e *Alloc*, indicando a presença dos defeitos relativamente rápido. O comportamento com o mutante *Sleep* necessitou de mais execuções para a detecção de anomalias, acusando a presença do defeito para três das cinco cargas de trabalho testadas. Isso indica que a presença do *sleep* teve pouco impacto no uso dos recursos aferidos.

Os resultados para a heurística restritiva são apresentados na Tabela 14, seguindo o mesmo código de cores da tabela anterior. Como a heurística restritiva espera duas acusações consecutivas da presença de defeitos, os resultados apresentados nas células são mostrados em pares.

Tabela 14 – Resultados do experimento da aplicação de criptografia - Heurística Restritiva

	Mutante			
Entrada	**BCLEAN**	**INFINITE**	**SLEEP**	**ALLOC**
Lowest	4-5	5-6	N	4-5
Low	3-4	7-8	N	3-4
Medium	3-4	4-5	14-15	4-5
High	4-5	4-5	N	3-4
Highest	3-4	3-4	25-26	4-5

Os resultados da heurística restritiva foram semelhantes aos resultados da heurística simples, com todos os casos positivos sendo encontrados na execução seguinte ao resultado anterior. A única diferença neste experimento foi um novo falso negativo para o mutante *Sleep* com a carga de trabalho *Low*, uma vez que a heurística simples acusou a presença do defeito apenas na última execução analisada.

As execuções da versão Controle têm seus resultados apresentados na Tabela 15. As células em azul representam os acertos da metodologia, onde não foi indicado um novo padrão de agrupamento e, assim, a metodologia não acusou a presença de defeitos. As células em vermelho destacam os erros da metodologia, onde foi acusada a presença de um defeito apesar de não terem sido inseridos defeitos no código do Controle.

Tabela 15 – Resultados do experimento da aplicação de criptografia - Controle

	CONTROLE	
Entrada	**HEURÍSTICA SIMPLES**	**HEURÍSTICA RESTRITIVA**
Lowest	28	28-29
Low	-	-
Medium	-	-
High	-	-
Highest	-	-

Apenas um caso de falso positivo foi identificado em ambas as heurísticas, para a entrada *Lowest* e após a análise da maior parte das execuções, mostrando que a metodologia necessita de uma alteração considerável para acusar uma anomalia. A anomalia encontrada para o mutante de controle pode ter sido causada por alguma alteração no sistema utilizado, apesar dos procedimentos realizados para minimizar esse risco, ou por alguma alteração natural na execução da aplicação.

6.6 Considerações finais

Os resultados apresentados neste capítulo mostram que o uso da metodologia *Tricorder* teve um resultado muito bom para a aplicação sob teste, com uma taxa de acerto de 88% para a heurística simples e de 84% para a heurística restritiva, mostrando que seria possível e também positivo seu uso como uma etapa de teste complementar dessa aplicação. Além do bom resultado geral, um bom resultado para o mutante *Controle* reforça os resultados para os mutantes defeituosos, mostrando que a metodologia detecta a presença de defeitos quando esses são executados, e não acusa, sistematicamente, a falsa presença de defeitos inexistentes.

A execução desse experimento foi necessária para um melhor entendimento da metodologia *Tricorder* e de todo o seu processo de uso, desde a escolha de uma aplicação com as características necessárias para a execução da etapa de monitoração até os ajustes realizados para otimizar os resultados da etapa de análise, como o uso de execuções de aquecimento e a definição da quantidade de execuções da aplicação original utilizadas.

CAPÍTULO

7

EXPERIMENTO ORDENAÇÃO EM JAVA

7.1 Considerações iniciais

O segundo experimento deste trabalho foi executado com o objetivo de testar a metodologia *Tricorder* com uma linguagem de programação diferente das linguagens já utilizadas. Buscou-se utilizar uma linguagem interpretada, com coletor de lixo e com um alto nível de relevância, chegando às opções Python e Java. Por já haver um trabalho em andamento no grupo de pesquisa que estava testando aplicações em Python, decidiu-se por utilizar a linguagem Java neste segundo experimento. Com a escolha de uma nova linguagem de programação, mantiveram-se as demais propriedades da aplicação, como alto uso de recursos e execução periódica.

A ordenação de números inteiros é um problema muito estudado em computação e bem conhecido, com métodos eficientes e com diferentes formas de implementá-los. O uso de métodos conhecidos de ordenação, em conjunto com uma linguagem de programação com suas próprias complexidades, possibilitou um foco maior no entendimento e exploração das características da linguagem para a *Tricorder*, contribuindo com a expansão do uso desta metodologia para aplicações em linguagens com as características de Java.

7.2 Aplicação

A aplicação *Multithreaded-Algorithms*, contendo exemplos de implementações de métodos paralelos de ordenação de números inteiros, foi escolhida a partir de pesquisas no Github (SHAOWDY, 2017). Desta aplicação, foram utilizados neste experimento os métodos de ordenação *Merge Sort* e *Quick Sort* em suas versões paralelas.

A aplicação executa um dos métodos de ordenação paralela, de acordo com a entrada utilizada. Na aplicação há um *loop* principal que escolhe e executa o método de ordenação de acordo com a entrada, por meio de uma estrutura *Switch*, tendo como opções os métodos *Merge*

Sort sequencial, *Merge Sort* paralelo e *Quick Sort* Paralelo. Em cada iteração do *loop* principal, o método de ordenação escolhido é aplicado em uma cópia da entrada.

7.3 Experimento

Foram utilizadas três métricas de uso de recursos neste experimento, as três métricas utilizadas no trabalho prévio (MONTES, 2019a), percentual de uso de CPU, uso de RAM (em MBytes) e quantidade de operações de Entrada e Saída. A monitoração do uso de recursos foi realizada utilizando o *script* Python pré-existente, com adaptações para a execução de uma nova aplicação (MONTES, 2019a).

A etapa de monitoração deste experimento seguiu o padrão definido para o experimento anterior, com 30 segundos de execução de onde são coletadas 300 amostras em intervalo de 100 milissegundos. Também foram mantidas as quantidades de execuções, com 50 execuções da aplicação original e 30 execuções dos mutantes.

Assim como a aplicação de criptografia, as entradas da aplicação de ordenação paralela também são arquivos de texto com números gerados aleatoriamente. Foi gerado um arquivo para cada nível de carga de trabalho *Low*, *Medium* e *High*, com 30, 60 e 100 milhões de números, respectivamente. Além da quantidade de números dos arquivos, as cargas de trabalho também variam o método de ordenação utilizado, com três cargas de trabalho que utilizam o *Merge Sort* paralelo e três cargas de trabalho que utilizam o *Quick Sort* paralelo, adicionando um pouco mais de complexidade ao experimento. As seis cargas de trabalho geradas são apresentadas na Tabela 16.

Tabela 16 – Cargas de trabalho definidas para a aplicação de ordenação em Java

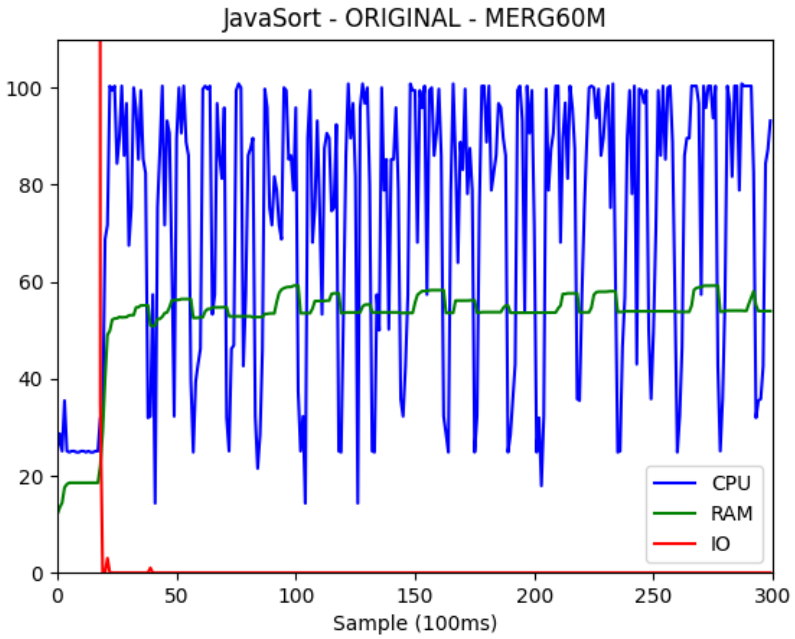
Carga de trabalho	Quantidade de números	Método de ordenação
MRGL	30 Milhões	MergeSort
MRGM	60 Milhões	MergeSort
MRGH	100 Milhões	MergeSort
QCKL	30Milhões	QuickSort
QCKM	60 Milhões	QuickSort
QCKH	100 Milhões	QuickSort

Os métodos de ordenação utilizam os recursos do computador de forma semelhante, com alto uso de CPU e um uso significativo de memória principal. De forma intuitiva e semelhante ao experimento anterior, o aumento da quantidade de números dos arquivos de entrada impactam diretamente no uso de recursos pela aplicação, com entradas maiores necessitando de mais tempo e mais memória para serem executadas.

7.3.1 Mutantes

Foram implementados quatro versões mutantes defeituosas neste experimento com a aplicação de ordenação paralela em Java. Foram utilizados apenas defeitos já conhecidos (descritos abaixo), mas adaptados para a implementação da aplicação testada. Além das versões defeituosas, ainda há o mutante de controle. A Figura 16 mostra o gráfico de uso de recursos da aplicação de ordenação em Java e do mutante de controle.

Figura 16 – Gráfico de uso de recursos da aplicação original

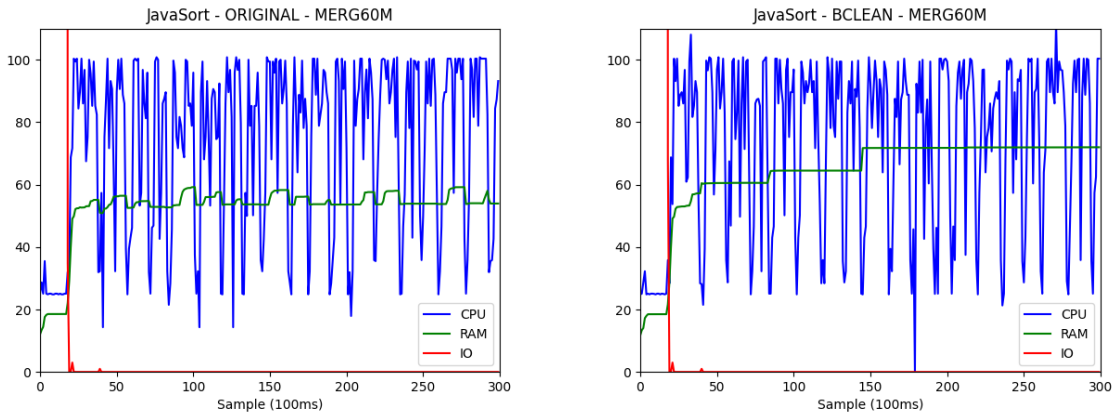


Fonte: Elaborada pelo autor.

A versão mutante *Bclean* foi implementada como uma condicional para uma execução do *Garbage Collector* após a ordenação, onde uma versão sem defeitos executa a limpeza de memória e o mutante *Bclean* não executa. Essa foi a adaptação utilizada para a utilização desse mutante na linguagem Java, com as versões da aplicação tendo o comportamento de uso de memória principal semelhante às aplicações implementadas em C/C++. A Figura 17 mostra o gráfico de uso de recursos do mutante *Bclean*, com o uso de memória crescente e se estabilizando, sem as limpezas do coletor de lixo.

Os mutantes *Sleep* e *Infinite* têm o mesmo comportamento do experimento anterior. O mutante *Infinite* foi implementado como um *loop* infinito e o mutante *Sleep* deixa a aplicação em espera por 200 milissegundos a partir do método *sleep* da classe *Thread*. Esses mutantes impactam no uso do processador pela aplicação, onde o *Infinite* monopoliza o uso de um núcleo do processador e o *Sleep* cria períodos onde o uso do processador chega a perto de 0% por alguns milissegundos. A Figura 18 mostra o gráfico de uso de recursos do mutante *Sleep*, onde é possível perceber os pontos onde o comando é executado, com as quedas do percentual de uso

Figura 17 – Execuções da aplicação original e aplicação com mutante *BCLEAN*

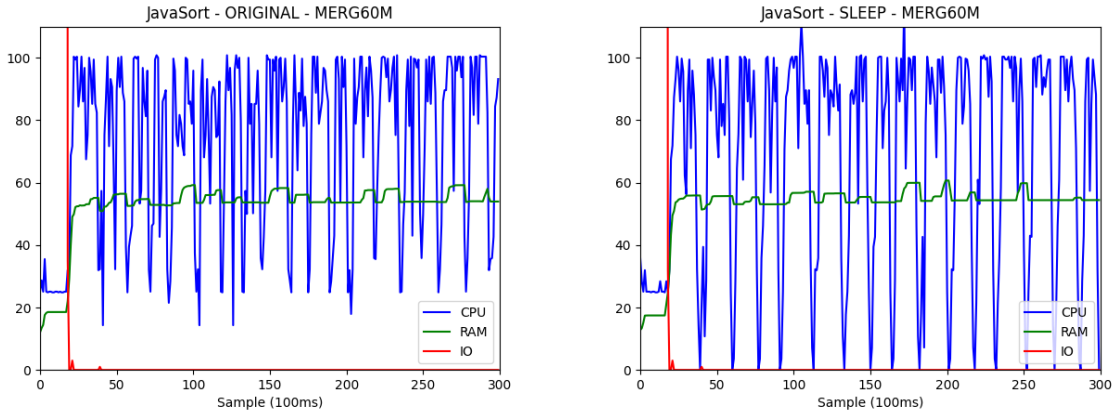


(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante BCLEAN

Fonte: Elaborada pelo autor.

de CPU. O gráfico de uso de recursos do mutante *Infinite* é apresentado na Figura 19, onde um núcleo do processador fica monopolizado pela execução do defeito.

Figura 18 – Execuções da aplicação original e aplicação com mutante *SLEEP*

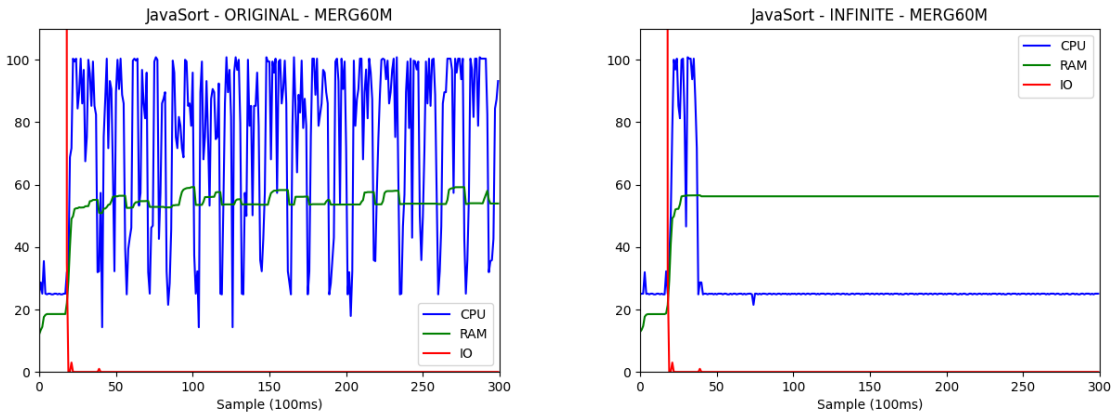


(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante SLEEP

Fonte: Elaborada pelo autor.

Aproveitando a característica da aplicação de executar métodos paralelos, foi implementado o mutante *Mono*. Este mutante tem o comportamento de serializar uma execução paralela, fazendo que uma aplicação que executaria em quatro núcleos do processador ao mesmo tempo execute em apenas um, com as *threads* competindo por seu uso. Esse mutante afeta o uso do processador pela aplicação ao limitar o número de núcleos do processador disponíveis para os métodos de ordenação. A Figura 20 mostra o gráfico de uso de recursos do mutante *Mono*, com a aplicação utilizando apenas um núcleo do processador, afetando todo o perfil da execução por

Figura 19 – Execuções da aplicação original e aplicação com mutante *INFINITE*

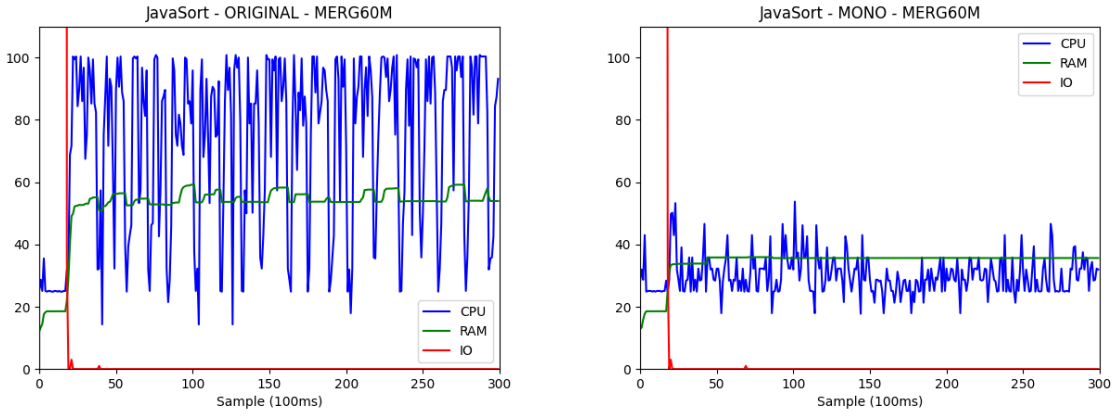


(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante INFINITE

Fonte: Elaborada pelo autor.

limitar o uso do processador para 25% de sua capacidade nas regiões paralelas afetadas pelo defeito.

Figura 20 – Execuções da aplicação original e aplicação com mutante *MONO*



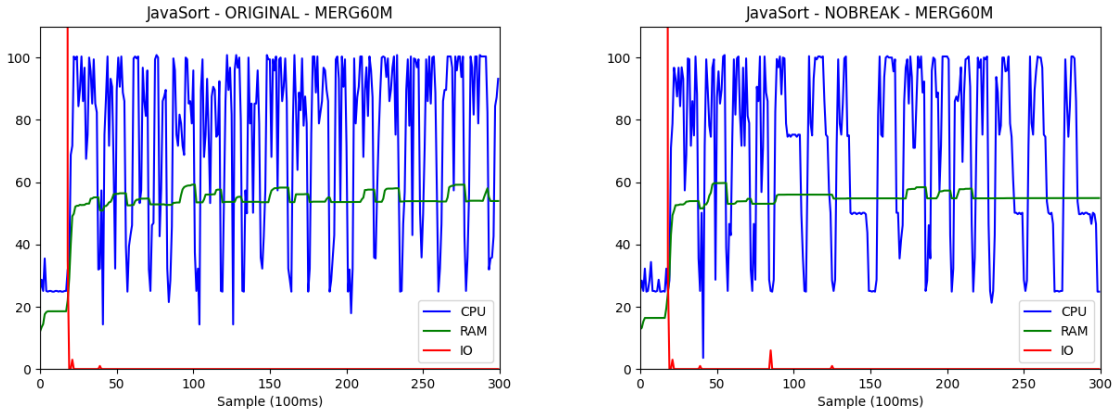
(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante MONO

Fonte: Elaborada pelo autor.

Para o quinto mutante utilizou-se o defeito *Nobreak*, aproveitando uma estrutura *Switch* existente na aplicação. A estrutura *Switch* é utilizada para executar o método de ordenação escolhido na entrada e o mutante *Nobreak* altera o funcionamento dessa estrutura retirando o comando *Break* após a execução do *Merge Sort*, com o método de ordenação *Quick Sort* sendo executado em seguida, desnecessariamente. Esse mutante altera o funcionamento da aplicação para as entradas que utilizam o *Merge Sort*, utilizando os recursos do computador de forma diferente. As entradas que utilizam o *Quick Sort* não são afetadas e executam normalmente a

aplicação. A Figura 21 mostra o gráfico de uso de recursos do mutante *Nobreak*, com o perfil da execução se misturando entre os dois tipos de carga de trabalho.

Figura 21 – Execuções da aplicação original e aplicação com mutante *NOBREAK*



(a) Uso de recursos da aplicação original

(b) Uso de recursos do mutante NOBREAK

Fonte: Elaborada pelo autor.

7.3.2 Resultados

Os resultados da aplicação da heurística simples para os dados de monitoração da aplicação de ordenação em java são exibidos na Tabela 17, onde as colunas representam as versões mutante com defeitos e as linhas representam as cargas de trabalho utilizadas. Seguindo o padrão já estabelecido, as células em amarelo indicam os casos onde a metodologia detectou corretamente a presença de um defeito e as células em vermelho indicam os casos onde um defeito foi executado mas sua presença não foi detectada pela metodologia, com os números indicando a quantidade de execuções analisadas até a detecção da anomalia. Para as entradas do tipo *QCK* o mutante *Nobreak* não executa nenhum defeito, com esses casos sendo representados pela cor azul na Tabela 17, indicando que nenhum defeito foi executado e a metodologia não detectou a presença de defeitos corretamente.

Tabela 17 – Resultados do experimento da aplicação de ordenação em Java - Heurística Simples

	Mutante				
Entrada	**BCLEAN**	**INFINITE**	**MONO**	**NOBREAK**	**SLEEP**
MRGL	5	7	2	3	5
MRGM	6	6	3	15	5
MRGH	6	3	4	28	11
QCKL	30	4	4	-	10
QCKM	11	5	3	-	10
QCKH	10	4	4	-	N

Esses resultados mostram uma alta eficácia da metodologia *Tricorder* para o teste dessa aplicação, indicando a presença de defeitos em quase todos os casos em que um defeito foi executado, com uma exceção, e não indicando a presença de defeitos quando defeito algum foi executado.

Tabela 18 – Resultados do experimento da aplicação de ordenação em Java - Heurística Restritiva

	Mutante				
Entrada	**BCLEAN**	**INFINITE**	**MONO**	**NOBREAK**	**SLEEP**
MRGL	5-6	7-8	2-3	3-4	7-8
MRGM	6-7	6-7	3-4	15-16	5-6
MRGH	6-7	3-4	4-5	N	11-12
QCKL	N	4-5	4-5	-	10-11
QCKM	21-22	5-6	3-4	-	10-11
QCKH	13-14	4-5	4-5	-	N

A Tabela 18 apresenta os resultados da aplicação da heurística restritiva. Os resultados modificaram um pouco em relação à heurística simples, com alguns casos onde foram necessárias mais execuções para a segunda detecção. Foram detectados dois novos casos de falsos negativos, como é esperado para a heurística restritiva, ocorrendo em casos onde a detecção na heurística simples ocorreu após a análise de um número alto de execuções e essa heurística necessita de mais execuções analisadas para gerar seus resultados.

Tabela 19 – Resultados do experimento da aplicação de ordenação em Java - Controle

	CONTROLE	
Entrada	**HEURÍSTICA SIMPLES**	**HEURÍSTICA RESTRITIVA**
MRGL	30	-
MRGM	-	-
MRGH	-	-
QCKL	-	-
QCKM	-	-
QCKH	-	-

Os resultados da aplicação das heurísticas simples e restritiva para a versão Controle são apresentados na Tabela 19. Seguindo o padrão, as células em azul indicam casos onde não foi executado um defeito e a metodologia não indicou a presença de defeitos e as células em vermelho indicam os casos de falso positivo, onde a metodologia indicou a presença de defeitos quando nenhum defeito foi executado. Para o mutante de controle houve apenas um caso de falso positivo utilizando a heurística simples, com essa detecção ocorrendo na última execução analisada, podendo ter sido causada por algum comportamento esperado do programa ou do sistema operacional.

7.4 Considerações finais

Apesar de alguns casos de falsos negativos e um caso de falso positivo, esse experimento obteve um dos melhores resultados da aplicação da metodologia *Tricorder*, com 94% de acerto da heurística simples e 91% de acerto para a heurística restritiva. Estes resultados mostram que a metodologia *Tricorder* tem um bom desempenho com aplicações semelhantes a uma aplicação de ordenação e que é possível o seu uso para testes de aplicações desenvolvidas em linguagens de programação como Java.

O teste de uma aplicação em Java, com todas as suas características que a diferem de uma aplicação em C, por exemplo, permite concluir que a metodologia *Tricorder* não está restrita a linguagens estruturadas, compiladas e com acesso mais direto à memória, dentre outras características. Também é possível sugerir que a metodologia *Tricorder* pode ser utilizada no teste de aplicações implementadas em outras linguagens que compartilham essas diferenças com o C.

CAPÍTULO

8

EXPERIMENTO OBS STUDIO

8.1 Considerações iniciais

No terceiro experimento buscou-se testar uma aplicação maior e mais complexa, com uma relevância já comprovada dentro de uma comunidade de usuários Além disso, as propriedades esperadas e descritas no Capítulo 5 foram mantidas, procurando por uma aplicação de execução escalável e com alto uso de recursos de sistema. De forma a complementar o trabalho realizado, buscou-se também que a aplicação escolhida utilizasse a placa de vídeo (GPU) no processamento, adicionando um novo recurso com duas novas métricas para a monitoração. A partir das buscas realizadas no Github, filtrando por projetos mais populares e utilizando um conhecimento prévio sobre as aplicações encontradas, foi escolhida a aplicação OBS Studio para o teste da metodologia.

O OBS Studio é um projeto complexo, composto por vários componentes e uma grande quantidade de arquivos de código fonte, resultando em uma aplicação com inúmeras configurações para os mais variados casos de uso. Essa é uma aplicação bem conhecida em seu nicho, que por sua vez possui uma demanda crescente de sua utilização em ambiente de trabalho, ensino ou entretenimento.

Apesar de realizar um trabalho que demanda muito do computador, o OBS Studio tem o objetivo de ser utilizado como uma ferramenta em segundo plano, enquanto uma outra aplicação de maior prioridade para o usuário está sendo executada. Essa característica é desafiadora para a *Tricorder*, pois a aplicação OBS Studio foi otimizada e projetada para reduzir o impacto no uso de recursos. A possibilidade de utilização da *Tricorder* como uma etapa de teste complementar para o OBS Studio é importante para o estudo desta metodologia de teste, pelo fato da aplicação ser, além das características já descritas, totalmente implementada por terceiros, possuir defeitos reais reportados, corrigidos e bem descritos.

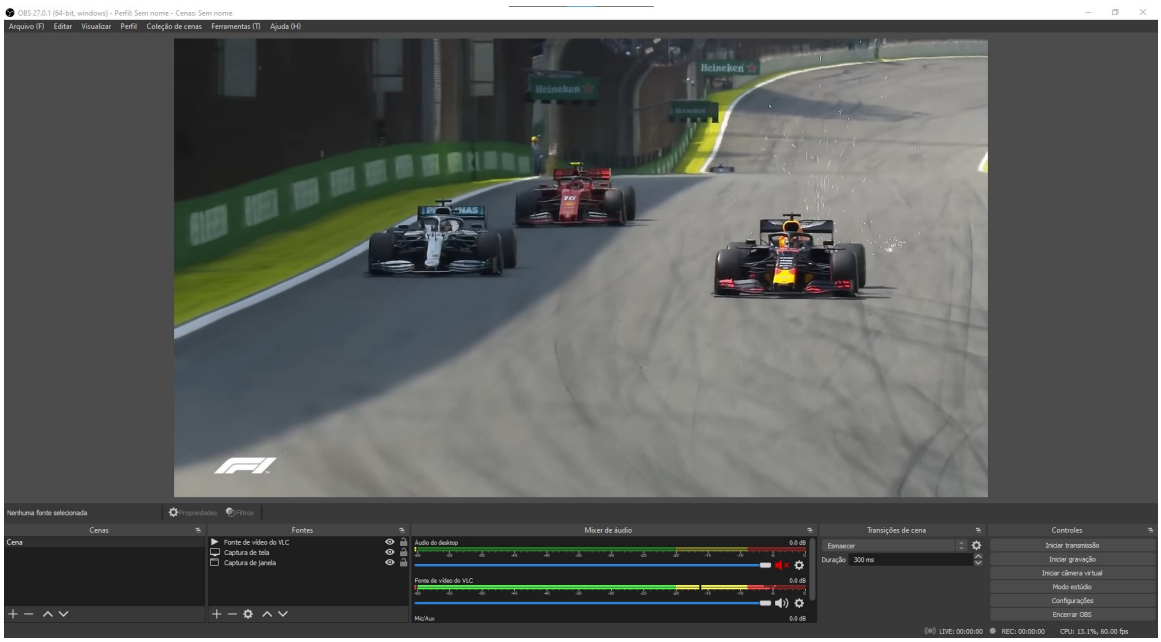
8.2 A aplicação OBS Studio

O OBS Studio (OBS, 2022) é uma aplicação de gravação e transmissão de vídeo muito utilizada para a gravação de vídeo aulas ou criação de conteúdo na internet, seja por gravação ou transmissão ao vivo de uma tela do computador.

O OBS tem diversas opções de funcionamento, com diversas configurações de saída, passando por formato e resolução de vídeo a diferentes codificadores de vídeo. Os primeiros experimentos com a *Tricorder* foram realizados com a gravação de um único vídeo executando externamente ao OBS, onde cada execução gravava um trecho diferente do vídeo. Esses experimentos preliminares tinham uma complexidade extra de iniciar o reprodutor de vídeo, além de necessitar de mais uma aplicação competindo pelo uso dos recursos do computador. Os resultados preliminares obtidos não foram satisfatórios, com a ocorrência de muitos falsos positivos. A partir de discussões com o grupo de pesquisa e estudo da aplicação, foram determinadas as opções descritas a seguir.

Para o funcionamento, foi escolhida a opção de gravação de um vídeo a partir de um arquivo, utilizando o reprodutor de mídia VLC (VIDEOLAN, 2022a) como fonte para a utilização no experimento, em conjunto com a opção de linha de comando de iniciar a gravação com a execução do programa, produzindo uma forma automática de execução da aplicação com a mesma entrada. Como entrada para o programa, foi utilizado um trecho de cerca de 40 segundos de um vídeo gravado com o próprio OBS, com resolução *Full HD* (1920x1080). O vídeo é executado pela opção de fonte de vídeo do VLC no OBS.

Figura 22 – Demonstração de uso do OBS Studio



Fonte: OBS (2022).

8.3 Experimento

Foram observadas seis métricas de uso de recursos durante a monitoração do OBS Studio. As métricas foram o percentual de uso de CPU, RAM (em MBytes), quantidade de operações de Entrada e Saída, e uso de Entrada e Saída (I/O) em Bytes. Foram adicionadas também a monitoração das métricas de uso de GPU e de uso de memória de vídeo (VRAM). As métricas relacionadas ao uso da placa de vídeo foram obtidas através da biblioteca *nvidia_-smi* (NVIDIA, 2022b) do *Python*, utilizada para obter dados relacionados ao estado atual de um dispositivo *NVIDIA GPU*. Essa biblioteca foi executada em conjunto à biblioteca *psutil* apresentada anteriormente, com as duas bibliotecas atuando sem interferir no funcionamento uma da outra.

A monitoração seguiu o padrão definido nos experimentos anteriores, com 50 execuções da aplicação original e 30 execuções dos mutantes. As amostras são extraídas a cada 100 milissegundos durante 30 segundos, totalizando 300 amostras de uso de recursos.

As cargas de trabalho utilizadas no experimento foram definidas a partir das configurações de saída do vídeo gravado. O OBS oferece duas opções de codificadores de vídeo, um codificador baseado em software e um codificador baseado em hardware. O codificador baseado em software utilizado é o *x264* (VIDEOLAN, 2022b), uma biblioteca de software livre utilizada para codificar vídeos para os formatos de compressão *H.264* e *MPEG-4* utilizando apenas o processador durante sua execução. O codificador baseado em hardware utilizado é o *NVENC* (NVIDIA, 2022a), que utiliza uma seção física de GPUs NVIDIA dedicada exclusivamente para a codificação sem afetar o desempenho da placa de vídeo ou do processador, cenário ideal para a gravação de jogos, renderizações ou outros casos onde se busca alto desempenho com um mínimo impacto no uso de recursos do computador.

Com os dois codificadores de vídeo oferecidos pelo OBS, foram definidas duas cargas de trabalho baseadas em Software e em Hardware. A partir desses dois tipos de carga, foram definidas outras três variações de carga, equivalentes cargas baixa, média e alta, com a escolha da resolução do vídeo de saída. As cargas baixas possuem a resolução 852x480 (480p), as cargas médias possuem a resolução 1280x720 (720p ou HD) e as cargas altas possuem a resolução 1920x1080 (1080p ou *Full HD*). Com os dois tipos de variações, foram definidas as seis cargas de trabalho descritas na Tabela 20.

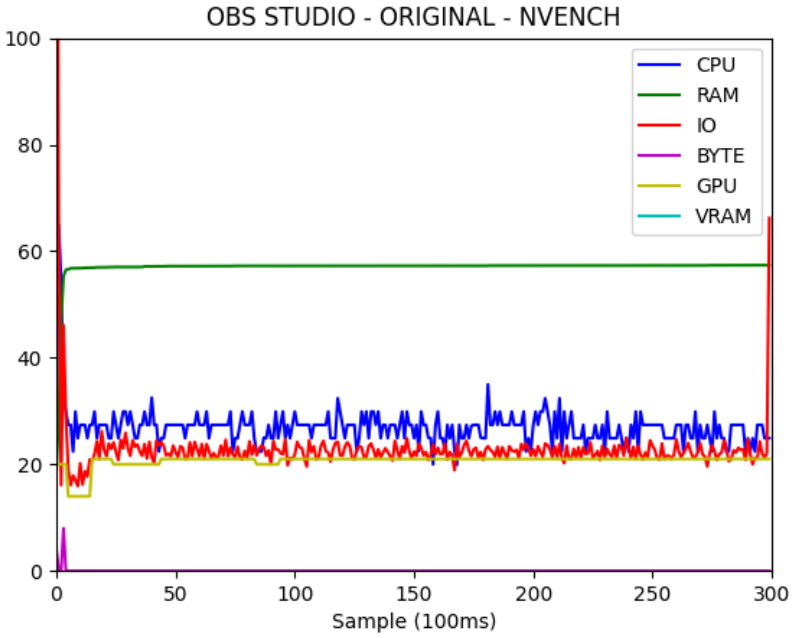
Tabela 20 – Cargas de trabalho definidas para a aplicação OBS Studio

Carga de trabalho	Resolução	Codificador de vídeo
NVENCL	480p	NVENC
NVENCM	720p	NVENC
NVENCH	1080p	NVENC
SOFTWL	480p	x264
SOFTWM	720p	x264
SOFTWH	1080p	x264

8.3.1 Mutantes

O uso do OBS Studio possui uma complexidade e importância extra para o teste da metodologia *Tricorder*, pois neste experimento foram utilizados mutantes baseados em defeitos reais reportados pelos programadores. As etapas de pesquisar, adaptar e implementar os defeitos a partir do repositório da aplicação foram as mais complexas de todo o processo do experimento, pela quantidade e diversidade de defeitos presentes no repositório, onde são poucos os defeitos com as características que permitem seu uso no experimento sem procedimentos muito complexos. Complementando o uso dos defeitos reais, também foram utilizados defeitos já utilizados em experimentos anteriores, além do mutante de controle sem defeitos. A Figura 23 mostra o gráfico de uso de recursos da aplicação OBS Studio e do mutante de controle.

Figura 23 – Gráfico de uso de recursos da aplicação original

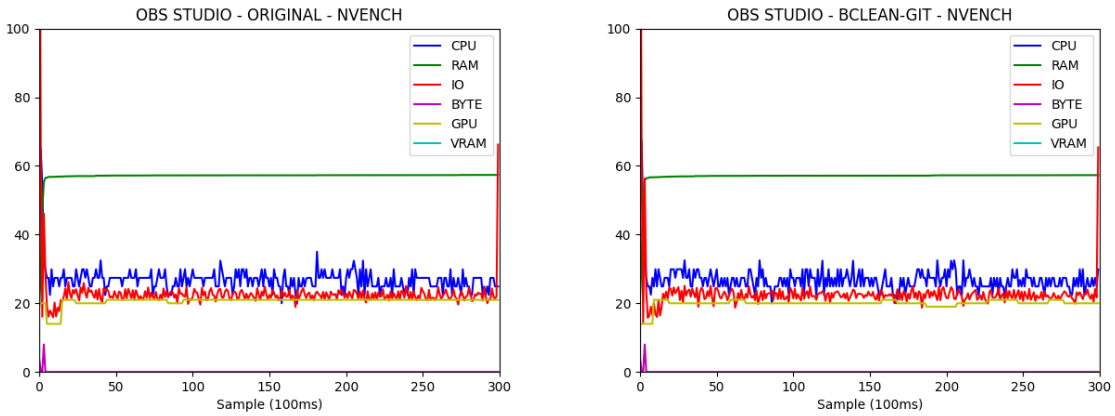


Fonte: Elaborada pelo autor.

O primeiro mutante implementado foi baseado em um defeito real da aplicação, obtido de uma *issue* (Issue #3306) no Github do projeto OBS Studio, o qual reporta um defeito já corrigido. O defeito reportado é um vazamento de memória (*memory leak*) detectado em uma região do código onde um *array* é alocado mas não é desalocado e outras desalocações são realizadas. Por coincidência, o defeito real reportado possui as mesmas características de um defeito já utilizado em outros experimentos, o *Bclean*, então manteve-se a nomenclatura para esse mutante, incluindo o sufixo "GIT"para indicar que é o mesmo defeito reportado pela *issue* do Github. A Figura 24 mostra o gráfico de uso de recursos do mutante *Bclean*.

Além da versão do defeito retirada do Github, também realizou-se uma modificação para aumentar os efeitos no uso de recursos em busca de melhorar a eficácia da detecção pela

Figura 24 – Execuções da aplicação original e aplicação com mutante *BCLEAN-GIT*

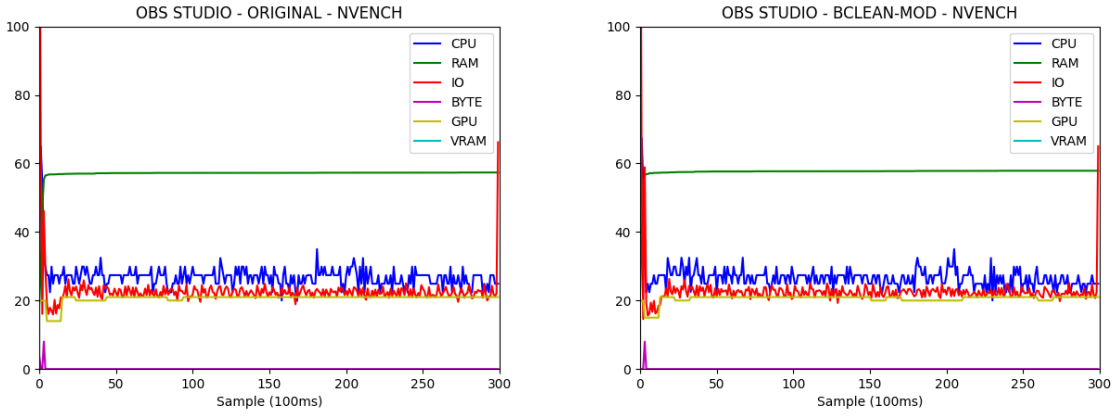


(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante BCLEAN-GIT

Fonte: Elaborada pelo autor.

metodologia. O defeito original deixa de desalocar dois *arrays* e a modificação realizada faz com que a aplicação deixe de desalocar uma quantidade maior de *arrays* além dos dois *arrays* do defeito original, resultando em um impacto maior no uso de memória pela aplicação. A modificação teve o efeito desejado, com pelo menos uma nova detecção pela metodologia. A Figura 25 mostra o gráfico de uso de recursos do mutante *Bclean* modificado.

Figura 25 – Execuções da aplicação original e aplicação com mutante *BCLEAN-MOD*



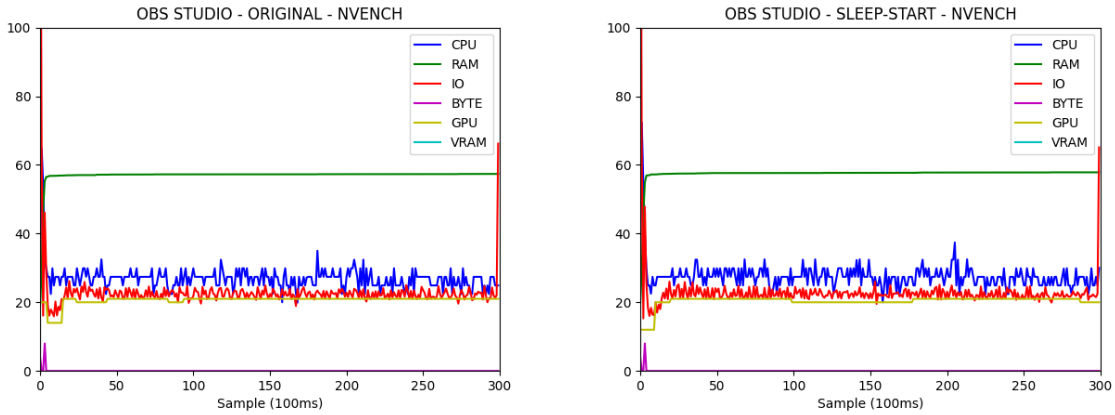
(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante BCLEAN-MOD

Fonte: Elaborada pelo autor.

Após utilizar um defeito real da aplicação, foi implementado o mutante *Sleep*, um defeito conhecido e utilizado em todos os experimentos. Para entender o impacto da posição no código onde o defeito está executando, foram implementadas duas versões desse mutante. A primeira versão possui o comando de parada da execução por 200 milissegundos no início da aplicação, onde é executada algumas vezes durante a fase da execução em que são aplicadas as

configurações e iniciada a gravação. Essa versão tem um efeito menor no uso de recursos e era esperado que a metodologia não tivesse um bom resultado na detecção do defeito. A Figura 26 mostra o gráfico de uso de recursos do mutante *Sleep*.

Figura 26 – Execuções da aplicação original e aplicação com mutante *SLEEP-START*

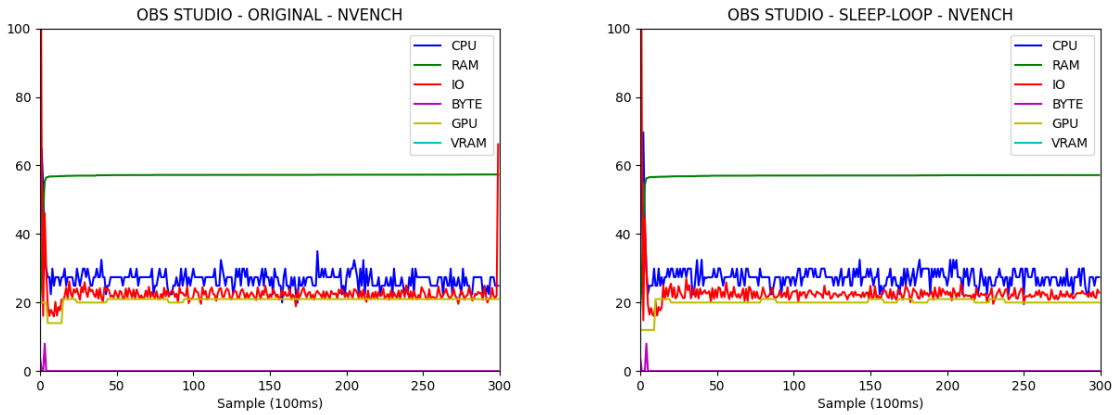


(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante SLEEP-START

Fonte: Elaborada pelo autor.

A segunda versão do *Sleep* foi implementada em uma estrutura de repetição, com o defeito sendo ativado durante toda a execução da aplicação. Esta versão afeta mais o uso de recursos e era esperado que tivesse um resultado melhor que a primeira versão do mutante *Sleep* neste experimento. A Figura 27 mostra o gráfico de uso de recursos desta segunda versão do mutante *Sleep*.

Figura 27 – Execuções da aplicação original e aplicação com mutante *SLEEP-LOOP*



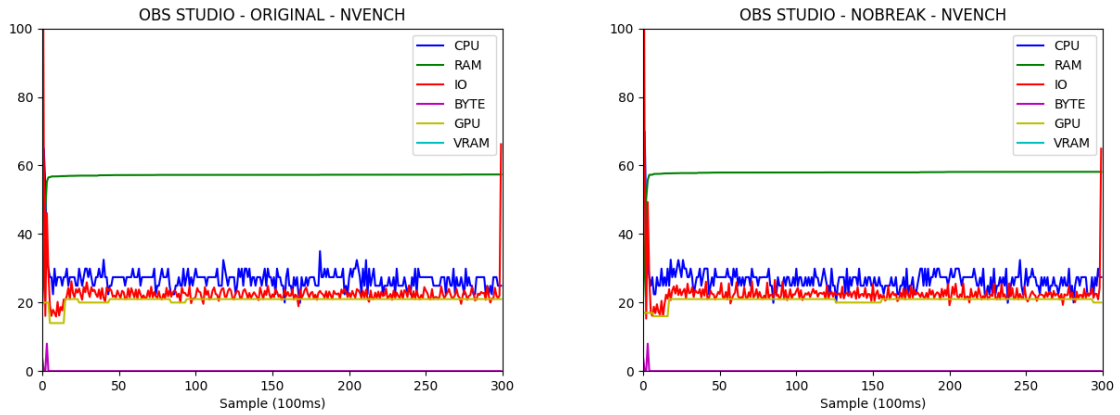
(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante SLEEP-LOOP

Fonte: Elaborada pelo autor.

Aproveitando uma estrutura *switch-case* encontrada no código, foi implementado o mutante *Nobreak*, onde um comando *break* foi deletado e o comportamento da aplicação alterado.

Este trecho de código é executado no início da aplicação e é responsável por chamar funções referentes às configurações de vídeo e imagem. A Figura 28 mostra o gráfico de uso de recursos do mutante *Nobreak*.

Figura 28 – Execuções da aplicação original e aplicação com mutante *NOBREAK*



(a) Uso de recursos da aplicação original

(b) Uso de recursos do mutante NOBREAK

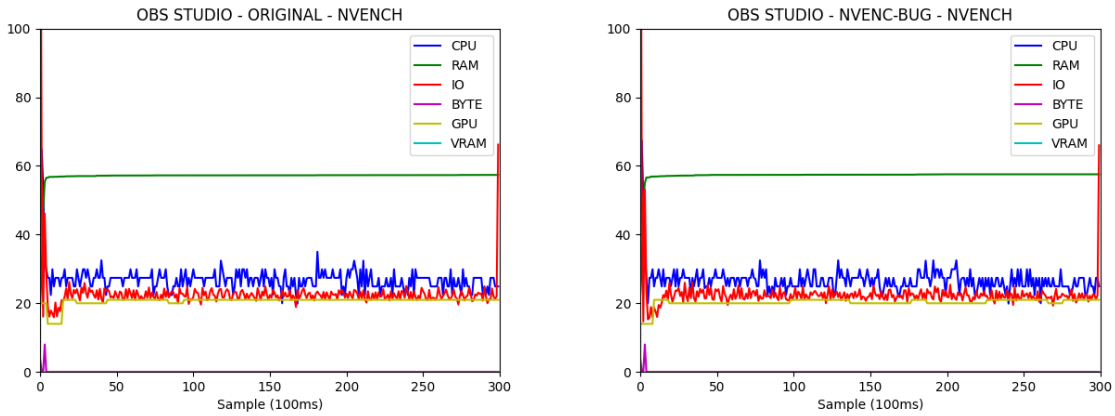
Fonte: Elaborada pelo autor.

O mutante *Nvenc-bug* foi baseado em um defeito real retirado do repositório do Github da aplicação, utilizando uma *issue* (Issue #4833) que reporta um defeito corrigido. Este defeito está relacionado ao uso da API do Nvenc, que estava sendo utilizada da maneira errada e foi corrigido para a utilização correta. O mutante modifica a implementação para a forma errada e o defeito somente é executado para cargas de trabalho que utilizam o codificador Nvenc, alterando o uso de recursos da GPU, com as cargas de trabalho que utilizam o codificador baseado em software executando normalmente. A Figura 29 mostra o gráfico de uso de recursos do mutante *Nvenc-bug*.

Utilizou-se mais um mutante baseado em defeito real. O mutante *Unlock* foi gerado a partir de uma *issue* do projeto do github do OBS Studio (Issue #2900), relacionada a um uso incorreto de *mutex* e espera de eventos do *Pthread*. Apesar de algumas linhas de código diferentes do mutante *Unlock* original, o efeito de sua execução é semelhante, então decidiu-se por manter o nome do mutante. O defeito reportado utiliza de forma equivocada comandos *wait* para sincronização de *threads* e um comando *unlock* para um *mutex* criado na mesma região de código, alterando a execução da aplicação e por consequência o seu uso de recursos. A Figura 30 mostra o gráfico de uso de recursos do mutante *Unlock*.

Todos os mutantes implementados para a aplicação OBS Studio resultaram em poucas alterações visíveis nos gráficos de uso de recursos, sendo que apenas o mutante *Nobreak* possui um pequeno aumento de uso de recursos no início da execução. Isso seria de grande dificuldade para um oráculo não automatizado (humano) de teste. Apesar disso, a *Tricorder* foi capaz de identificar as anomalias na maioria dos casos, como será apresentado nesta próxima seção.

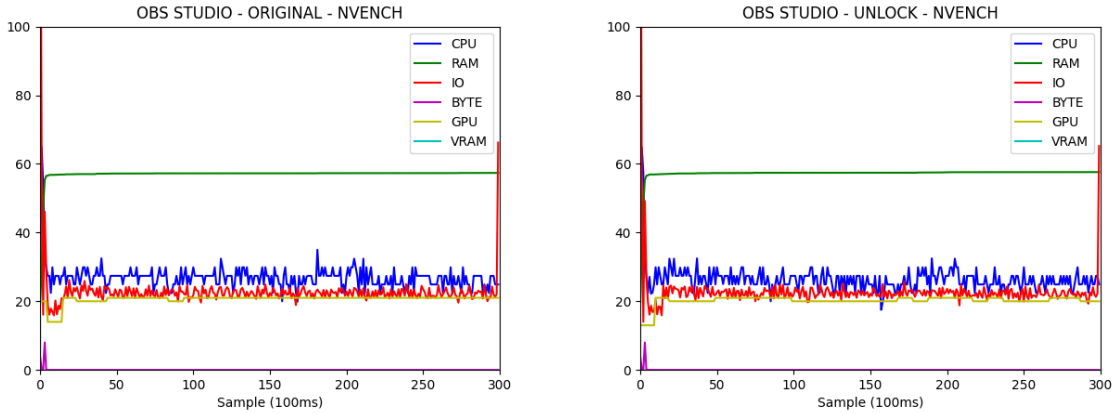
Figura 29 – Execuções da aplicação original e aplicação com mutante *NVENC-BUG*



(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante NVENC-BUG

Fonte: Elaborada pelo autor.

Figura 30 – Execuções da aplicação original e aplicação com mutante *UNLOCK*



(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante UNLOCK

Fonte: Elaborada pelo autor.

8.3.2 Resultados

O uso da heurística simples tem seus resultados apresentados nas Tabelas 21 e 22, seguindo o padrão dos demais experimentos, com as linhas representando as cargas de trabalho e as colunas representando as versões mutantes utilizadas. As células em amarelo indicam os casos onde a metodologia acusou a presença de um defeito que existia, as células em vermelho indicam os casos onde a metodologia não indicou a presença de um defeito executado e as células em azul indicam os casos onde nenhum defeito foi executado e a metodologia não indicou a presença de defeitos. Ainda seguindo o padrão, os números nas células indicam quantas execuções foram analisadas até a primeira detecção de uma anomalia.

Os resultados do uso da heurística restritiva são apresentados na Tabela 23, seguindo o

Tabela 21 – Resultados do experimento da aplicação OBS Studio - Heurística Simples

	Mutante			
Entrada	**BCLEAN-GIT**	**BCLEAN-MOD**	**SLEEP-START**	**SLEEP-LOOP**
NVENCL	N	12	N	15
NVENCM	11	13	26	6
NVENCH	5	17	10	10
SOFTWL	N	N	14	N
SOFTWM	24	N	25	15
SOFTWH	N	25	23	20

Tabela 22 – Resultados do experimento da aplicação OBS Studio - Heurística Simples

	Mutante		
Entrada	**NOBREAK**	**NVENC-BUG**	**UNLOCK**
NVENCL	13	24	N
NVENCM	22	22	10
NVENCH	9	6	11
SOFTWL	22	-	12
SOFTWM	17	-	13
SOFTWH	16	-	13

mesmo padrão de cores e números, onde o par de números indica em quais execuções foram encontradas anomalias duas vezes seguidas. Os resultados foram semelhantes aos resultados da heurística simples, com dois novos casos de falsos negativos e alguns casos onde a segunda detecção não ocorreu logo após a execução onde foi detectado o defeito na heurística simples.

Tabela 23 – Resultados do experimento da aplicação OBS Studio - Heurística Restritiva

	Mutante			
Entrada	**BCLEAN-GIT**	**BCLEAN-MOD**	**SLEEP-START**	**SLEEP-LOOP**
NVENCL	N	12-13	N	15-16
NVENCM	11-12	13-14	26-27	6-7
NVENCH	5-6	21-22	10-11	10-11
SOFTWL	N	N	14-15	N
SOFTWM	24-25	N	N	N
SOFTWH	N	25-26	28-29	20-21

A análise do mutante de controle tem seus resultados apresentados na Tabela 25. A tabela segue o padrão de cores dos demais experimentos, com as células em vermelho indicando detecções de anomalias em casos onde não havia um defeito sendo executado e as células em azul indicando os casos onde nenhum defeito foi executado e a metodologia não indicou a presença de defeitos.

A análise das execuções do mutante de controle resultaram em um caso de falso positivo que se repetiu em ambas as heurísticas após a análise de 20 execuções, o que não é suficiente para comprometer o experimento. Uma execução normal da aplicação ainda pode apresenta uma

Tabela 24 – Resultados do experimento da aplicação OBS Studio - Heurística Restritiva

	Mutante		
Entrada	**NOBREAK**	**NVENC-BUG**	**UNLOCK**
NVENCL	13-14	24-25	N
NVENCM	22-23	22-23	10-11
NVENCH	11-12	11-12	11-12
SOFTWL	22-23	-	12-13
SOFTWM	19-20	-	13-14
SOFTWH	16-17	-	13-14

Tabela 25 – Resultados do experimento da aplicação OBS Studio - Controle

	CONTROLE	
Entrada	**HEURÍSTICA SIMPLES**	**HEURÍSTICA RESTRITIVA**
NVENCH	-	-
NVENCM	20	20-21
NVENCL	-	-
SOFTWL	-	-
SOFTWM	-	-
SOFTWH	-	-

diferença no uso de recursos que pode ser identificado pela metodologia como uma anomalia, assim como a execução de alguma tarefa do Sistema Operacional ou algum fator físico podem afetar de alguma forma a execução da aplicação.

8.4 Considerações finais

Apesar de alguns poucos falsos positivos e de alguns falsos negativos na análise dos mutantes defeituosos, a metodologia *Tricorder* foi capaz de identificar corretamente a presença ou não de defeitos em 81% dos casos analisados com a heurística simples e em 77% dos casos para a heurística restritiva.

As variações dos mutantes *Bclean* e *Sleep* tiveram um resultado bom, porém com muitos falsos negativos, de certa, já esperados em função da posição em que se encontravam no código fonte. Mesmo assim, o uso da *Tricorder* ainda foi capaz de indicar a presença de defeitos em alguns desses casos, mesmo com baixos impactos no uso dos recursos.

O uso dos mutantes *Bclean*, *Nvenc-bug* e *Unlock* mostram que é possível utilizar a metodologia *Tricorder* para detectar defeitos reais em aplicações reais, uma vez que os defeitos foram reportados por desenvolvedores da aplicação.

CAPÍTULO

9

EXPERIMENTO SIMULAÇÃO DE FLUIDOS

9.1 Considerações iniciais

O quarto experimento deste trabalho foi realizado com o teste de aplicações voltadas para as simulações computacionais, pelo seu consumo de recursos do computador, e seu uso e relevância em diversas áreas e aplicações. Dentre as diversas opções de simulações computacionais disponíveis no Github, foi encontrada uma aplicação que implementa métodos de simulações de fluidos.

Simulações computacionais de fluidos são essenciais para áreas da engenharia e da física e possuem diversas aplicações, como previsão e estudos sobre o clima, design de veículos dentre outros (KOCHKOV *et al.*, 2021). Seus algoritmos normalmente consideram problemas complexos que exigem bastante do hardware e do software básico para se obter simulações realistas em tempos aceitáveis.

9.2 A aplicação de simulação de fluidos

O projeto *Jet framework* (KIM, 2022) contém as implementações apresentadas em (KIM, 2017), e fornece uma base para o uso de diversos métodos de simulação de fluidos, com exemplos de uso dos métodos implementados. Ao instalar e compilar o código, são gerados executáveis relacionados a cada um dos métodos implementados, possibilitando a execução de um método de simulação de fluidos específico e a escolha de um de seus exemplos de uso por meio de argumentos de linha de comando, onde também podem ser fornecidos parâmetros de configuração da simulação que será executada.

O método de simulação de fluidos *Smoothed Particles Hydrodynamics* (SPH) foi o escolhido para a execução do experimento. O método SPH é um método Lagrangiano de modelagem sem malha de interação de partículas de fluidos e superfícies, com vantagens para a

modelagem de problemas mais complexos de simulações de fluidos (LIU; ZHANG, 2019).

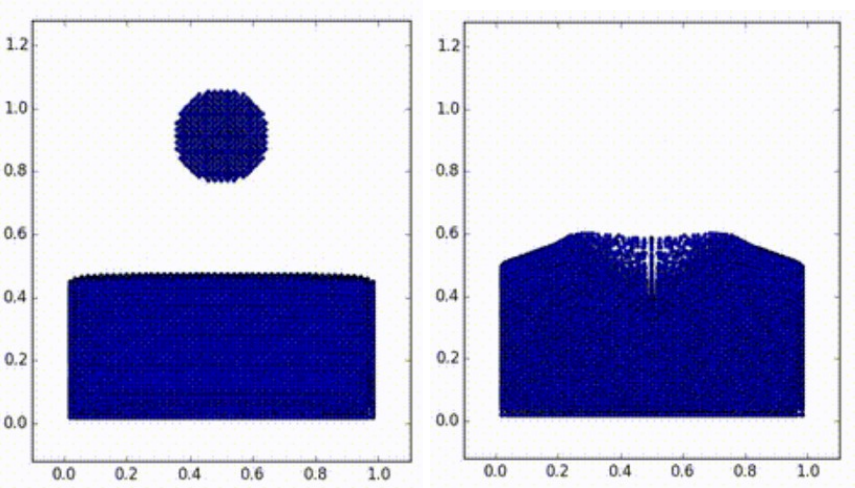
9.3 Experimento de simulação de fluidos

A monitoração da aplicação de simulação de fluidos foi executada observando quatro métricas de uso de recursos percentual de CPU, uso de RAM (em MBytes), quantidade de operações de E/S e uso de E/S (em Bytes), já apresentadas em experimentos anteriores. A aplicação *Jet framework* é executada apenas no processador sem a utilização da GPU, sendo desnecessário o uso de métricas relacionadas à placa de vídeo.

Seguindo os experimentos anteriores, foram monitoradas 50 execuções da aplicação original e 30 execuções de cada mutante. O tempo de monitoração segue com 30 segundos de execução com um intervalo de 100 milissegundos entre cada amostra, resultando em 300 amostras de uso de recursos.

A implementação do método *SPH* de simulação de fluidos apresenta três exemplos de uso em sistemas tridimensionais, duas simulações de queda de corpo de água e uma simulação de quebra de barragem. A simulação de queda de um corpo de água gera uma esfera de partículas acima de um outro corpo do líquido em repouso e simula a ação da gravidade sobre este sistema, simulando a queda de uma gota de água e os efeitos causados em todo o sistema. Dois *frames*, em momentos distintos desta simulação, são apresentados na Figura 31.

Figura 31 – Frames da simulação do exemplo *WaterDrop*



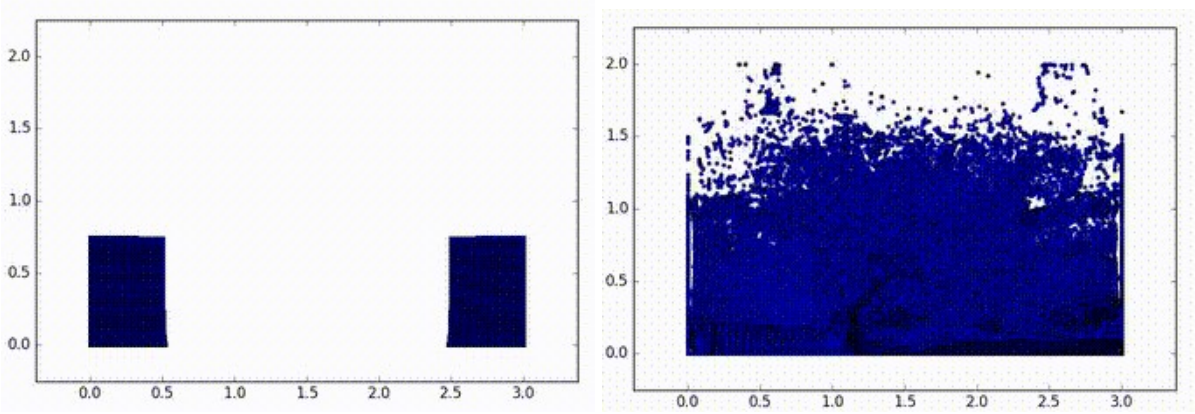
(a) Início da simulação (b) Simulação executando

Fonte: Kim (2022).

O exemplo de quebra de barreira cria dois corpos de partículas nas extremidades de um sistema retangular, simulando duas barragens de água que sofrem uma ruptura. Após a queda das barragens, os corpos de partículas seguem para o centro do ambiente da simulação, onde

se chocam entre si e com três barreiras cilíndricas, simula-se a interação das partículas com o ambiente. Dois *frames* em momentos distintos desta simulação são apresentados na Figura 32.

Figura 32 – Frames da simulação do exemplo *DamBreaking*



(a) Início da simulação (b) Simulação executando

Fonte: Kim (2022).

As cargas de trabalho escolhidas para o experimento utilizam dois dos três exemplos implementados, sendo um dos exemplos a queda de corpo de água e outro o exemplo de quebra de barragem de forma a ter alguma variação do uso de recursos durante a execução. Além dos exemplos executados também foi variado o parâmetro de espaçamento entre as partículas, onde um espaçamento maior resulta em uma quantidade menor de partículas e, por consequência, um tempo menor de execução a cada iteração do método. Estes parâmetros são fornecidos por argumentos da linha de comando, com o código utilizando alguns valores padrões caso não sejam fornecidos. O valor padrão para o espaçamento entre partículas sugerido pelo autor no código é de 0.20 unidades e, por este motivo, decidiu-se manter tal valor como uma das entradas utilizadas. Este valor padrão de 0.20 unidades é pesado o suficiente para executar apenas uma iteração do *loop* principal da aplicação. Assim, ficou definido como parte das cargas com o maior uso de recursos. Após testes com diferentes variações do parâmetro de espaçamento entre partículas e análise da variação do uso de recursos, foi decidida a utilização dos valores 0.40 e 0.60 unidades para as entradas média e baixa, seguindo a característica inversamente proporcional entre o valor e uso de recursos. As seis cargas de trabalho geradas a partir da combinação entre o exemplo executado e o espaçamento entre partículas estão listadas na Tabela 26.

9.3.1 Mutantes

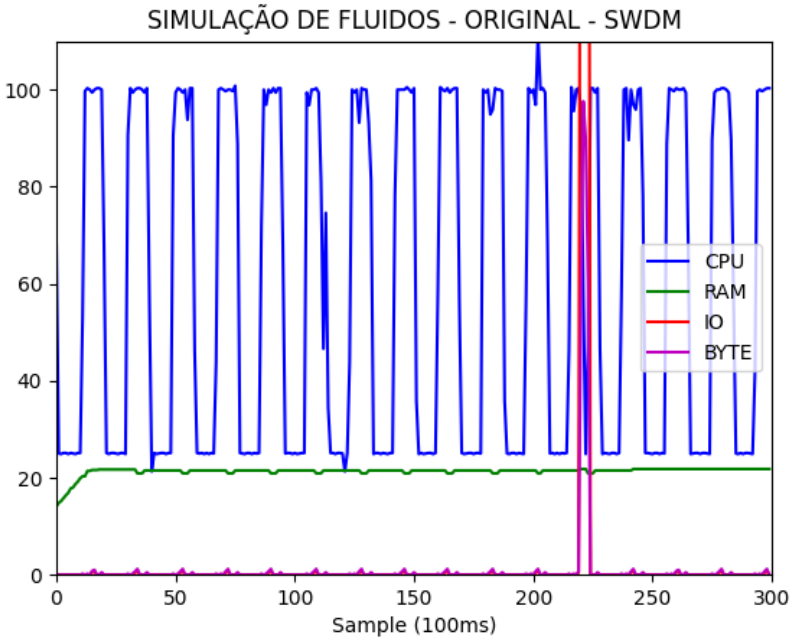
Para este experimento foram utilizados defeitos já conhecidos e novos, implementados a partir da análise do código e de discussões com o grupo de pesquisa. Apesar do projeto do Github ter algumas discussões entre usuários, não foram encontrados defeitos com as características necessárias para a execução do experimento. Assim como os demais experimentos, o mutante de

Tabela 26 – Cargas de trabalho definidas para a aplicação simulação de fluidos

Carga de trabalho	Espaçamento	Método
SWDL	0,020	SPH - Water Drop
SWDM	0,040	SPH - Water Drop
SWDH	0,060	SPH - Water Drop
SDBL	0,020	SPH - Dam Breaking
SDBM	0,040	SPH - Dam Breaking
SDBH	0,060	SPH - Dam Breaking

controle (idêntico ao algoritmo original) foi utilizado. A Figura 33 mostra o gráfico de uso de recursos da aplicação de simulação de fluidos e do mutante de controle.

Figura 33 – Gráfico de uso de recursos da aplicação original

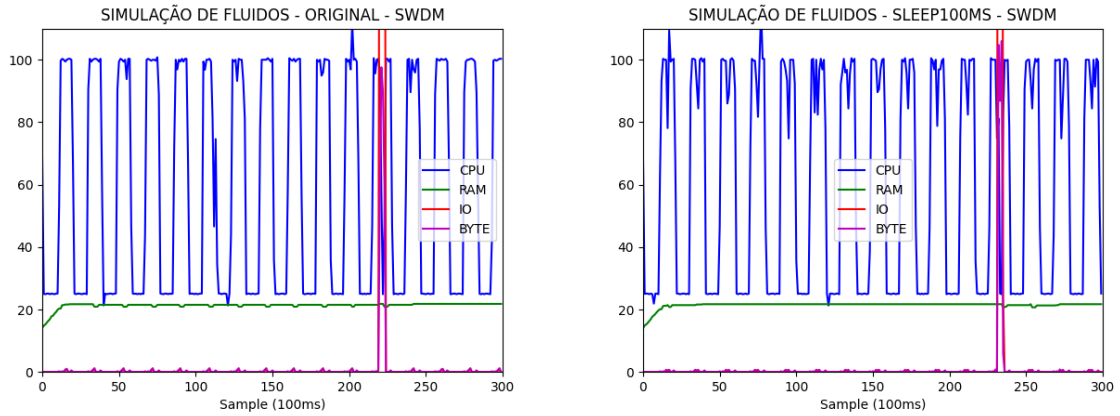


Fonte: Elaborada pelo autor.

O mutante *Sleep* foi utilizado novamente, executando um comando para parar a execução durante um determinado período de tempo. Primeiramente foi implementada uma versão com um intervalo de 100 milissegundos durante a execução e esta versão foi utilizada como um teste inicial do uso da metodologia com a aplicação, sem esperar um resultado muito bom. A Figura 34 mostra o gráfico de uso de recursos do mutante *Sleep* com 100 ms de intervalo, sendo possível observar alterações nos trechos onde a aplicação demanda muito do processador.

Após os primeiros testes com o *Sleep*, foi implementada uma nova versão com um intervalo maior, com uma parada de 200 milissegundos na execução, onde o objetivo era aumentar o impacto do defeito no uso de recursos da aplicação. Era esperado que a metodologia detectasse mais anomalias nas execuções ou que estas anomalias fossem detectadas com a análise

Figura 34 – Execuções da aplicação original e aplicação com mutante *SLEEP100MS*

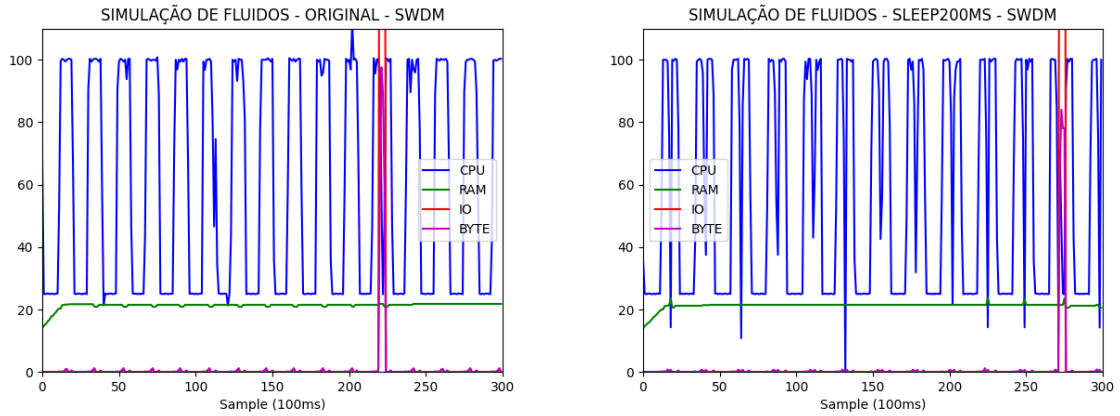


(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante SLEEP100MS

Fonte: Elaborada pelo autor.

de um número menor de execuções. A Figura 35 mostra o gráfico de uso de recursos do mutante *Sleep* com 200 ms de intervalo, com os pontos de execução do comando se destacando mais que a versão anterior.

Figura 35 – Execuções da aplicação original e aplicação com mutante *SLEEP200MS*



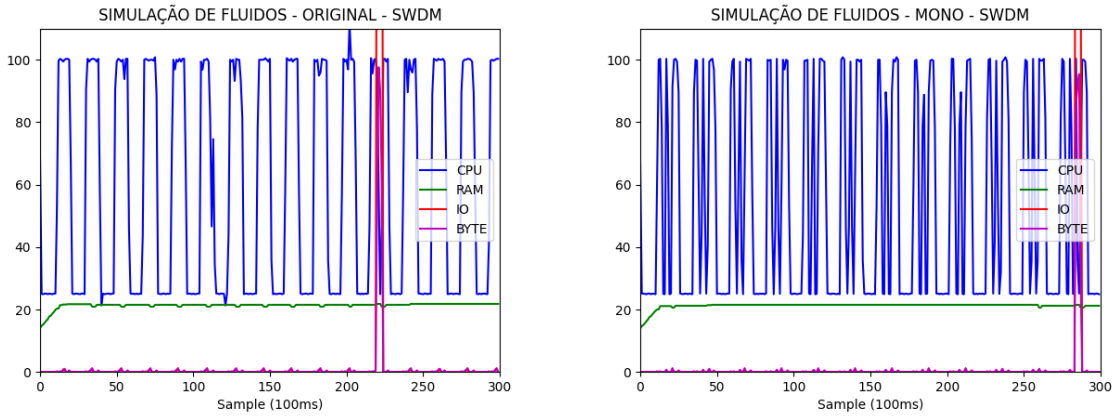
(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante SLEEP200MS

Fonte: Elaborada pelo autor.

A aplicação testada neste experimento possui regiões paralelas em seu código, onde são executadas iterações com algum processamento paralelo em *arrays* com quantidade de elementos proporcional à quantidade de partículas simuladas. Com esta característica da aplicação, foi implementada uma versão do mutante *Mono* focada em uma destas regiões paralelas, onde uma estrutura *for* paralela foi substituída por uma estrutura *for* sequencial. Este defeito inserido afeta principalmente o uso de CPU ao limitá-lo proporcionalmente à quantidade de núcleos do

processador disponíveis. A Figura 36 mostra o gráfico de uso de recursos do mutante *Mono*, onde há trechos em que o uso do processador é limitado a um único núcleo.

Figura 36 – Execuções da aplicação original e aplicação com mutante *MONO*



(a) Uso de recursos da aplicação original

(b) Uso de recursos do mutante MONO

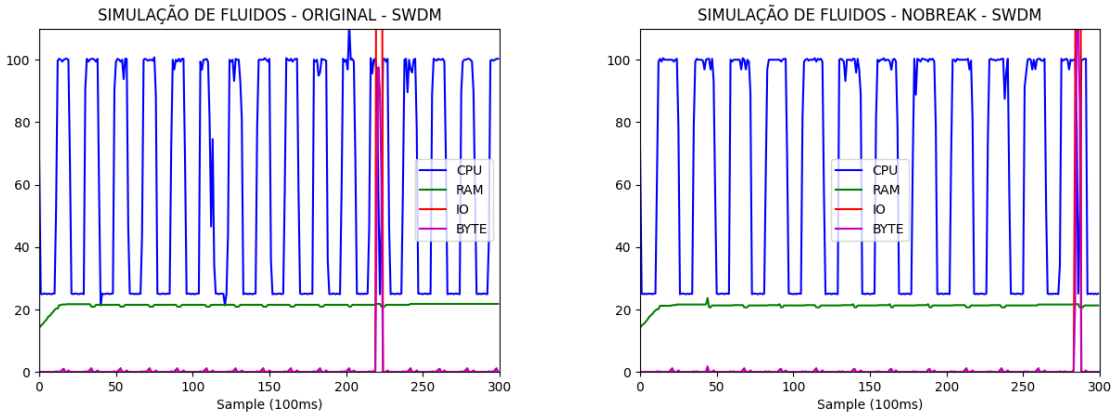
Fonte: Elaborada pelo autor.

O mutante *Nobreak* é outro defeito já utilizado em experimentos prévios que foi adaptado para esta aplicação. Para aproveitar particularidades da aplicação e diversificar os defeitos estudados, a implementação deste mutante foi realizada de forma diferente. Em um dos laços de repetição executados na aplicação há uma verificação de uma taxa de erro máxima aceitável, onde a execução do *loop* é interrompida quando um limite é atingido por meio de um comando *break*. A implementação do defeito retira a verificação e a interrupção do *loop*, permitindo que a estrutura de repetição siga até o fim das iterações. A Figura 37 mostra o gráfico de uso de recursos do mutante *Nobreak*, com um maior intervalo de alto uso do processador como efeito da modificação do código.

Um defeito de lógica foi inserido na aplicação para a implementação do mutante *Logic*. O cálculo de uma variável de controle foi alterado para uma fração inversa, onde um intervalo de tempo é calculado ao dividir o tempo restante pelo número de passos e passou a ser calculado dividindo o número de passos pelo tempo restante, resultando em um valor diferente do esperado e afetando a execução da aplicação. Com a execução deste defeito, a aplicação de simulação de fluidos muda sua forma de execução e gera resultados incorretos porém ainda válidos, alterando o tempo gasto entre cada iteração e geração de um novo quadro da simulação. A Figura 38 mostra o gráfico de uso de recursos do mutante *Logic*, com um perfil de uso de recursos muito diferente do perfil de uso de recursos da aplicação original.

As discussões com o grupo de pesquisa resultaram em um novo mutante. Ao analisar um trecho do código fonte da aplicação enquanto era discutida a implementação de outro defeito, foi sugerida a modificação de um *Vector* adicionando o modificador *Static*, uma situação já experienciada anteriormente pela equipe. Isso altera o comportamento e o uso de recursos da

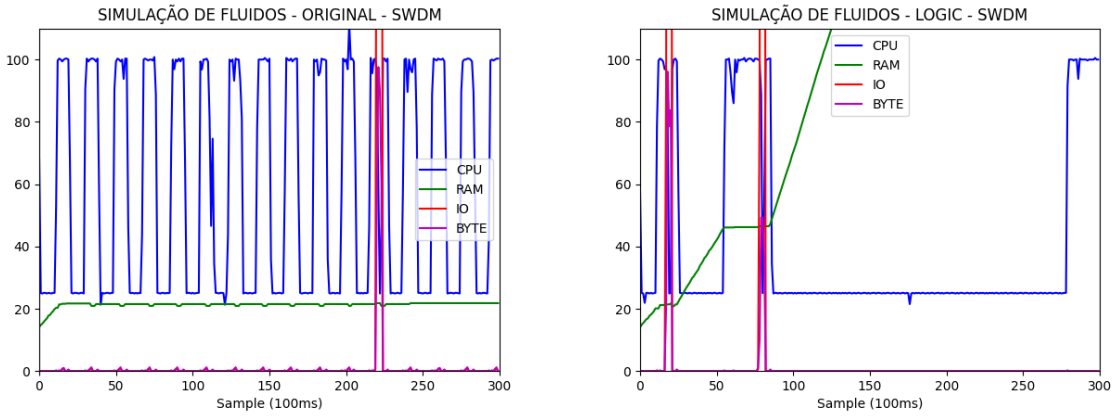
Figura 37 – Execuções da aplicação original e aplicação com mutante *NOBREAK*



(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante NOBREAK

Fonte: Elaborada pelo autor.

Figura 38 – Execuções da aplicação original e aplicação com mutante *LOGIC*



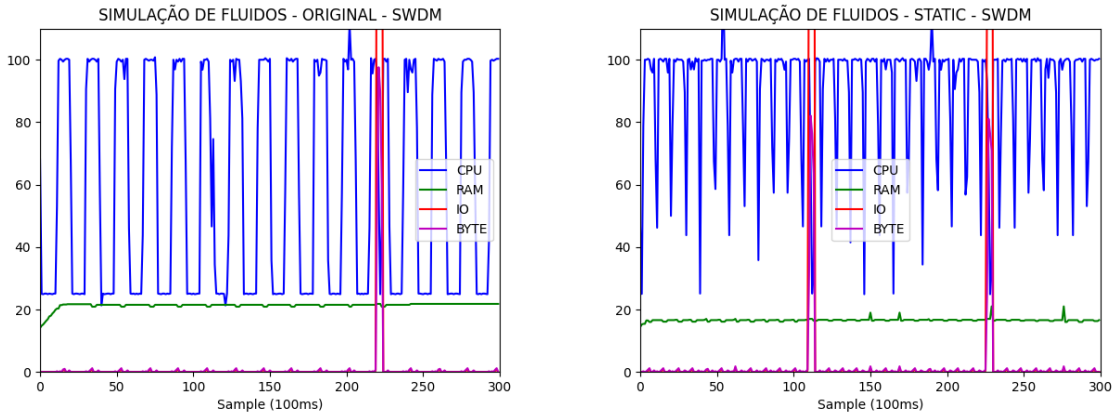
(a) Uso de recursos da aplicação original (b) Uso de recursos do mutante LOGIC

Fonte: Elaborada pelo autor.

aplicação. A execução desta versão defeituosa também resulta em respostas dentro da faixa válida de valores, porém, incorretas, o que dificulta sua detecção apenas com a análise do resultado gerado. A Figura 39 mostra o gráfico de uso de recursos do mutante *Static*, onde os *loops* da execução são executados em um espaço de tempo menor, impactando em toda a duração da monitoração.

Foi implementado um novo mutante com o objetivo de testar um defeito de sincronização. A intenção deste mutante é afetar o bloqueio de uma variável compartilhada, algo semelhante ao mutante *Unlock* utilizado em experimentos anteriores. No entanto, neste caso a aplicação implementa o compartilhamento de variáveis em seções paralelas de forma implícita, sem o uso de *mutex*. Como forma de estudar melhor o comportamento da metodologia, decidimos

Figura 39 – Execuções da aplicação original e aplicação com mutante *STATIC*



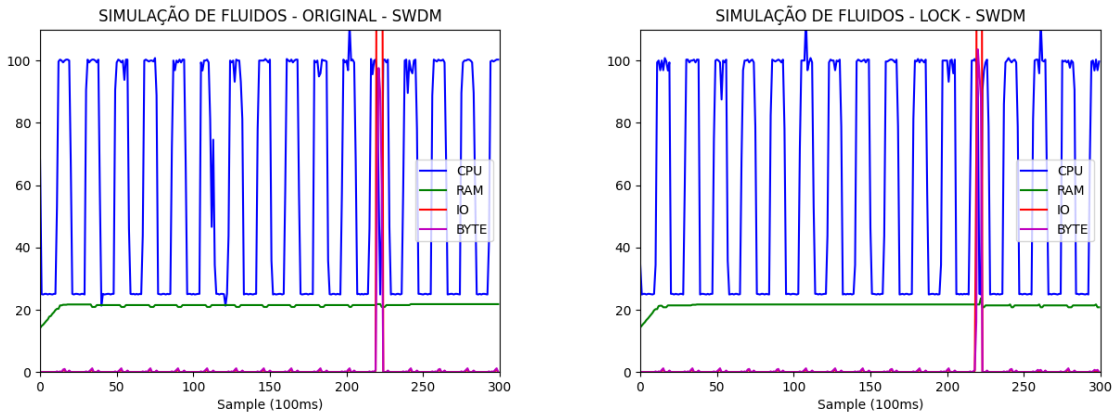
(a) Uso de recursos da aplicação original

(b) Uso de recursos do mutante STATIC

Fonte: Elaborada pelo autor.

implementar um defeito com a característica contrária, onde uma variável compartilhada passou a ter um bloqueio explícito por meio de um *mutex* com uma estrutura *lock*. Este mutante foi nomeado de *Lock* e afeta pouco o uso de recursos do computador, sendo esperado que a metodologia tenha dificuldades em detectar sua presença durante a análise. A Figura 40 mostra o gráfico de uso de recursos do mutante *Unlock*, com alterações imperceptíveis para um observador.

Figura 40 – Execuções da aplicação original e aplicação com mutante *LOCK*



(a) Uso de recursos da aplicação original

(b) Uso de recursos do mutante LOCK

Fonte: Elaborada pelo autor.

9.3.2 Resultados

A Tabela 27 apresenta os resultados do uso da heurística simples para a análise do experimento com a aplicação de simulação de fluidos. As cores das células seguem o padrão

já estabelecido nos demais experimentos deste trabalho, com as células em amarelo indicando acertos da metodologia e as células em vermelho indicando os falsos negativos. Os números nas células indicam a quantidade de execuções analisadas até a detecção da primeira anomalia, indicando a presença de um defeito.

Tabela 27 – Resultados do experimento da aplicação de simulação de fluidos - Heurística Simples

	Mutante						
Entrada	**LOCK**	**LOGIC**	**MONO**	**NOBREAK**	**SLEEP2**	**SLEEP1**	**STATIC**
SWDL	30	9	4	6	3	3	5
SWDM	N	6	20	29	6	26	8
SWDH	N	10	11	26	N	N	4
SDBL	N	4	14	23	12	19	12
SDBM	N	5	17	19	25	17	7
SDBH	26	N	9	15	9	27	14

Os resultados da aplicação da heurística restritiva são apresentados na Tabela 28, seguindo o padrão de cores já citados. Esta heurística acusa a presença de um defeito caso sejam encontradas anomalias duas vezes seguidas, com estas execuções sendo representadas nos números das células. Houve três novos casos de falsos negativos, um deles para o mutante *Lock*, com alguns casos onde a segunda anomalia não ocorreu imediatamente após a detecção da heurística simples.

Tabela 28 – Resultados do experimento da aplicação de simulação de fluidos - Heurística Restritiva

	Mutante						
Entrada	**LOCK**	**LOGIC**	**MONO**	**NOBREAK**	**SLEEP2**	**SLEEP1**	**STATIC**
SWDL	N	9-10	4-5	6-7	3-4	3-4	5-6
SWDM	N	6-7	20-21	29-30	6-7	26-27	8-9
SWDH	N	10-11	11-12	26-27	N	N	4-5
SDBL	N	4-5	16-17	N	12-13	19-20	12-13
SDBM	N	7-8	17-18	23-24	28-29	17-18	7-8
SDBH	26-27	N	13-14	21-22	9-10	N	14-15

Os resultados da análise do mutante de controle são apresentados na Tabela 29, com as células em azul representando casos onde não foi executado nenhum defeito conhecido e a metodologia não indicou a presença de nenhum defeito. Neste experimento não ocorreu nenhum caso de falso positivo, onde a metodologia indica a presença de um defeito onde nenhum foi executado.

9.4 Considerações finais

Os resultados do uso da metodologia *Tricorder* para o teste desta aplicação foram bons, de acordo com o esperado para os mutantes utilizados. O mutante *Lock* afeta pouco o uso de

Tabela 29 – Resultados do experimento da aplicação de simulação de fluidos - Controle

	CONTROLE	
Entrada	**HEURÍSTICA SIMPLES**	**HEURÍSTICA RESTRITIVA**
SWDL	-	-
SWDM	-	-
SWDH	-	-
SDBL	-	-
SDBM	-	-
SDBH	-	-

recursos pela aplicação, não sendo esperado que fosse detectada a sua presença ou que esta detecção fosse difícil, o que é possível observar nos resultados. Apesar disso, a metodologia foi capaz de identificar a anomalia em dois casos para a heurística simples e em um caso para a heurística restritiva.

Os resultados para os demais mutantes foram muito bons, apesar de alguns falsos negativos. O uso dos mutantes conhecidos apresentou bons resultados, com a modificação do mutante *Nobreak* mantendo um resultado satisfatório. O uso da heurística simples resultou em uma taxa de acerto de 85%, enquanto a heurística restritiva foi capaz de detectar a presença ou não de defeitos em 79% dos casos, mostrando uma boa eficiência do uso da metodologia *Tricorder* no teste de aplicações com as características do nicho testado.

Com este experimento foi possível ampliar o teste de aplicações reais e utilizar alguns defeitos novos, testando também a capacidade da metodologia de encontrar defeitos com pouco impacto no desempenho em alguns casos. Foi possível também consolidar o processo de uso da metodologia *Tricorder*, passando pelas etapas de escolha e posterior estudo da aplicação, definição dos defeitos utilizados e adaptações das cargas de trabalho e *scripts* para as etapas de monitoração e de análise.

CAPÍTULO

10

CONCLUSÃO

10.1 Análise dos resultados

Este trabalho tem como objetivo estender a metodologia *Tricorder* (MONTES, 2019a) quanto à sua eficácia no teste de aplicações desenvolvidas para distintos nichos de atuação.

Para cumprir este objetivo, a eficácia da *Tricorder* foi avaliada por meio de experimentos que executaram a metodologia com quatro diferentes aplicações, todas representativas em seus respectivos segmentos de atuação: criptografia de arquivos, ordenação de números inteiros, gravação de vídeos e simulação de fluidos. Além dos testes com as novas aplicações, também foram utilizadas novas métricas de desempenho, com as adaptações necessárias para seu uso, e novos defeitos foram inseridos nas aplicações para a geração de versões mutantes.

Foram utilizadas aplicações desenvolvidas por terceiros e disponíveis publicamente em repositórios na web. Diferentes defeitos foram inseridos nestas aplicações, sendo que a inserção considerou os resultados de pesquisas disponíveis na literatura sobre taxonomias de defeitos. Foram usados nos experimentos, sempre que possíveis, defeitos reais descritos nestas aplicações escolhidas.

Em um caso específico, durante os testes da aplicação de criptografia do primeiro experimento, a *Tricorder* foi capaz de revelar de fato um defeito real de vazamento de memória(*memory leak*), o qual não era de conhecimento dos seus autores ainda. Este defeito pertencia à classe *Bclean*, já usada neste projeto para se testar a eficácia da metodologia. Durante os testes de uma versão do próprio *Bclean*, percebemos que o uso de memória continuava aumentando e foi possível identificar uma variável que era alocada mas não era liberada ao fim do *loop* principal da aplicação, que são as características do mutante *Bclean*.

Um outro resultado importante foi a eficácia do uso da *Tricorder* para revelar defeitos reais da aplicação OBS Studio. Apesar de representarem situações reais, os defeitos inseridos nas outras aplicações ainda são definidos e inseridos manualmente, sendo seu impacto conhecido

ou com indicações de seu resultado. O uso de defeitos reais, apesar de alguns casos de defeitos semelhantes aos já utilizados, mostram um caso real de uso da metodologia *Tricorder*, onde uma aplicação implementada por terceiros possui defeitos reais, cujas presenças são detectadas pela metodologia, sem interferências diretas do testador.

A metodologia *Tricorder* foi capaz de identificar corretamente a presença dos defeitos em mais de 82% dos casos testados nos experimentos, como é possível observar nas Tabelas 30 e 31. A partir destes resultados, é possível concluir que a metodologia *Tricorder* pôde ser utilizada com eficácia nos experimentos realizados em aplicações voltadas a diferentes nichos de atuação, como uma etapa de testes complementar às técnicas pré-existentes. O teste de aplicações em diferentes nichos foi possível com adaptações para a geração dos dados de monitoração, como a definição de cargas de trabalho e automatização de sua execução. Isso indica que a metodologia ainda requer um refinamento da sua automatização, embora o núcleo da metodologia, o agrupamento dos dados de monitoração, tenha se mostrado constantemente eficaz para o teste proposto na metodologia.

Tabela 30 – Resultados da aplicação da metodologia - Heurística Simples

	Simples				
Aplicação->	**Cripto**	**Javasort**	**OBS**	**SimSph**	**Total**
Acertos	88,00%	94,44%	81,25%	85,42%	86,62%
Erros	12,00%	5,56%	18,75%	14,58%	13,38%

Tabela 31 – Resultados da aplicação da metodologia - Heurística Restritiva

	Restritiva				
Aplicação->	**Cripto**	**Javasort**	**OBS**	**SimSph**	**Total**
Acertos	84,00%	91,67%	77,08%	79,17%	82,17%
Erros	16,00%	8,33%	22,92%	20,83%	17,83%

Apesar das adaptações citadas anteriormente, foram testadas com sucesso aplicações de terceiros implementadas nas linguagens C, C++ e Java, e executadas nos sistema operacionais *Windows* e *Linux*, sem a necessidade de adaptações nos *scripts* de monitoração ou de análise para as mudanças de plataforma.

10.2 Ameaças à validade

Alguns pontos podem ser considerados como ameaças à validade dos resultados obtidos, sendo o fator humano uma das principais causas. A maior parte das decisões tomadas durante a execução do projeto, principalmente as mais impactantes como a escolha das aplicações, defeitos e cargas de trabalho, foram discutidas em reuniões com o grupo de pesquisa que é o mesmo grupo que propôs a metodologia. A discussão em grupo, com revisão dos procedimentos e com a participação dos integrantes na escolha das aplicações, métricas e defeitos ameniza bastante

essa ameaça mas ainda pode haver um viés mesmo que involuntário nessas decisões, apesar de nossos esforços para evitar essa situação.

Uma das maiores dificuldades encontradas na execução do trabalho foi a definição de quais defeitos utilizar para implementar os mutantes, principalmente encontrar defeitos reais nos repositórios das aplicações. Outro aspecto, ainda relacionado aos defeitos, é a localização dos mesmos nos códigos analisados. A localização dos defeitos nos códigos afeta de maneira diferente o uso dos recursos computacionais e, assim, impacta a eficácia da *Tricorder*. Com esta dificuldade em pauta, foram pesquisadas na literatura disponível taxonomias de defeitos que norteassem nossas escolhas de defeito. Além disso, foram utilizados defeitos já conhecidos e testados em experimentos anteriores pelo grupo na Tricorder. Esta abordagem para a escolha dos defeitos auxiliou bastante na execução dos experimentos e nas configurações para o uso de cada aplicação.

Mesmo considerando os nichos de atuação distintos, o número de aplicações testadas ainda é pequeno e isso pode ser considerada uma ameaça. Foi possível testar aplicações com características distintas, utilizar novas métricas e aprimorar as configurações de uso da metodologia, mas por questões de tempo e da complexidade das aplicações não foi possível realizar um número maior de experimentos.

10.3 Desafios

Os principais desafios encontrados no desenvolvimento deste trabalho estão relacionados às aplicações utilizadas nos experimentos. A definição dos nichos de aplicações e a sugestão de alguns exemplos em cada nicho não foram tarefas muito complexas, ao contrário da análise dessas aplicações em busca das características necessárias para a execução dos experimentos. As aplicações escolhidas devem permitir o uso de um *script* para sua execução, possibilitar o uso de algumas cargas de trabalho diferentes e fornecer suporte para sua instalação e uso.

Com as aplicações definidas, as dificuldades encontradas foram relacionadas em entender o funcionamento de cada aplicação estudando seu código fonte para a inserção de defeitos. O objetivo do trabalho era a utilização de aplicações de terceiros e essas dificuldades já eram esperadas e foram contornadas, cumprindo o objetivo do trabalho.

Encontrar defeitos reais nos repositórios das aplicações foi uma tarefa difícil, já que as aplicações que possuem um projeto mais estruturado e de colaboração pública são mais complexas. O OBS Studio foi a única aplicação utilizada nos experimentos que possui um projeto de onde foi possível utilizar defeitos reportados pelos desenvolvedores.

Modificar os *scripts* de monitoração para utilizar métricas relacionadas ao uso de GPU foi um desafio também. A biblioteca *psutil* não fornece suporte a essas métricas e a primeira biblioteca testada para esse fim impactou a execução do *psutil*, alterando a monitoração das

métricas já usadas. Utilizando mais uma biblioteca de monitoração foi possível obter as métricas necessárias da forma correta, sem impactar a monitoração das outras métricas.

10.4 Contribuições

A principal contribuição deste trabalho foi verificar a abrangência de aplicações que podem ser testadas com a metodologia *Tricorder*, utilizando para tanto, quatro aplicações de diferentes nichos. A partir das aplicações testadas, algumas outras contribuições podem ser citadas. Acredita-se que esta contribuição permitirá o amadurecimento da *Tricorder* como uma metodologia de teste complementar a técnicas já existentes e focada principalmente em softwares que encontram-se em execução e que necessitam de constantes manutenções.

Foram utilizadas aplicações que executavam em sistemas operacionais diferentes, com duas aplicações executando no Sistema Operacional *Windows 10* e duas aplicações executando no Sistema Operacional *Linux*, na distribuição Ubuntu 20.04. Os *scripts* de monitoração foram utilizados nos dois sistemas com as mesmas bibliotecas, sem a necessidade de modificações para cada plataforma.

O segundo experimento testou uma aplicação implementada em Java, uma linguagem com comportamento diferente das linguagens C e C++ das demais aplicações. Uma aplicação em Java executa em uma Máquina Virtual Java (*JVM*), com outras características importantes, como o coletor de lixo e o processo de compilação do código.

Pelo menos três defeitos reais foram utilizados no experimento do OBS Studio, além de uma modificação de um dos defeitos. O teste dessa aplicação mostrou que a metodologia pode ser usada para o teste de aplicações reais implementadas por terceiros para detectar defeitos reais encontrados em outras etapas de teste.

Além dos defeitos reais e dos defeitos já conhecidos de outros experimentos, novos defeitos e novas versões de defeitos conhecidos foram implementados e testados, aumentando a quantidade e variação de defeitos analisados pela metodologia *Tricorder*. De forma a guiar a definição desses defeitos, foi gerada uma taxonomia simples de defeitos a partir do estudo de taxonomias de defeitos da literatura, formando um guia para a escolha de defeitos utilizados nos experimentos desse trabalho e em possíveis experimentos de trabalhos futuros.

A monitoração de métricas relacionadas ao uso de GPU e um guia de instalação das ferramentas utilizadas pela metodologia *Tricorder* foram contribuições coletivas do grupo de pesquisa. O uso de métricas de GPU também era necessário em outro trabalho do grupo, onde as bibliotecas utilizadas e as modificações realizadas foram escolhidas e discutidas em grupo. O guia de instalação das ferramentas foi desenvolvido de forma colaborativa para auxiliar o uso dessas ferramentas nesses trabalhos em execução e para os próximos trabalhos que vão necessitar de passar por essa etapa, já com exemplos de sua utilização.

10.5 Trabalhos futuros

A partir do trabalho executado, surgiram dúvidas e questionamentos que podem inspirar o desenvolvimento de novos trabalhos. Os testes de mais aplicações de diferentes nichos, com novos defeitos, novas linguagens e novas métricas expandiram a abrangência do projeto da metodologia *Tricorder*, fornecendo mais dados e experiências para a investigação dedicada em diferentes pontos da metodologia.

O estudo dos defeitos precisa ser feito sob a perspectiva da *Tricorder*, verificando quais defeitos são possíveis dela detectar e quais não são, ortogonalmente à aplicação. A definição dos defeitos para o teste de cada aplicação foi um dos principais desafios do trabalho, sendo realizada de forma empírica a partir de algum conhecimento prévio da aplicação, da linguagem ou dos defeitos já utilizados. Embora tenham sido utilizados alguns defeitos reais, a análise do repositório de defeitos da aplicação foi uma tarefa trabalhosa e com uma baixa taxa de defeitos selecionados em relação aos analisados. Um estudo sobre os defeitos reais de uma aplicação como o OBS Studio pode contribuir para um melhor entendimento da metodologia, indicando seus pontos fortes e suas limitações em relação aos defeitos utilizados.

Foram utilizadas novas métricas de uso de recursos neste trabalho, mas não foram realizados estudos mais aprofundados sobre as métricas utilizadas e seus efeitos na metodologia. Alguns testes foram realizados alterando quais métricas seriam utilizadas na análise dos dados de monitoração e sobre representação dessas métricas nos arquivos de monitoração. Novos trabalhos podem focar no estudo das métricas utilizadas e de novas métricas disponíveis nas bibliotecas que foram utilizadas ou em novas bibliotecas, investigando o impacto das métricas na monitoração e na análise dos dados gerados.

Alguns trabalhos no projeto da metodologia *Tricorder* já estão em andamento. Um trabalho de mestrado foi realizado paralelamente a este trabalho, onde o foco foi o uso da metodologia *Tricorder* para o teste de aplicações de aprendizagem de máquina, investigando a eficácia da metodologia na detecção de defeitos relacionados a esse nicho. Outro trabalho paralelo realizado foi um trabalho de conclusão de curso focado no estudo dos parâmetros da *Damicore*, com o objetivo de melhorar os resultados obtidos com uso da ferramenta e diminuir o tempo gasto na etapa de agrupamento e análise. Dois novos trabalhos de mestrado estão em andamento, onde um deles pesquisará sobre novas ferramentas de agrupamento para comparar com o desempenho da *Damicore* e o outro irá estudar os processos da etapa de agrupamento buscando paralelizar este processo com o objetivo de otimizá-lo.

REFERÊNCIAS

ALAM, K. A.; AHMAD, R.; KO, K. Enabling far-edge analytics: performance profiling of frequent pattern mining algorithms. **IEEE Access**, IEEE, v. 5, p. 8236–8249, 2017. Citado na página 30.

ANUKEN. **Mindustry**: The automation tower defense game. 2022. Disponível em: <https://github.com/Anuken/Mindustry>. Citado na página 68.

ARORA, N.; BELL, J.; IVANČIĆ, F.; KAISER, G.; RAY, B. Replay without recording of production bugs for service oriented applications. In: **Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering**. [S.l.: s.n.], 2018. p. 452–463. Citado na página 30.

ATTARIYAN, M.; CHOW, M.; FLINN, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In: . [s.n.], 2012. p. 307–320. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84979001459&partnerID=40&md5=6c1a0632620d11d0a1ac40e85e6d81c3>. Citado nas páginas 22 e 26.

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. **IEEE transactions on dependable and secure computing**, IEEE, v. 1, n. 1, p. 11–33, 2004. Citado nas páginas 15, 55 e 56.

BANERJEE, A.; SRIVASTAVA, A. A cloud performance analytics framework to support online performance diagnosis and monitoring tools. In: **Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering**. [S.l.: s.n.], 2019. p. 151–158. Citado na página 30.

BARHAM, P.; DONNELLY, A.; ISAACS, R.; MORTIER, R. Using magpie for request extraction and workload modelling. In: **OSDI**. [S.l.: s.n.], 2004. v. 4, p. 18–18. Citado na página 25.

BERKHIN, P. A survey of clustering data mining techniques. In: **Grouping multidimensional data**. [S.l.]: Springer, 2006. p. 25–71. Citado na página 37.

BHATTACHARYYA, A.; AMZA, C. Pret: A tool for automatic phase-based regression testing. In: IEEE. **2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)**. [S.l.], 2018. p. 284–289. Citado nas páginas 28 e 30.

CASOLA, V.; BENEDICTIS, A. D.; RAK, M.; VILLANO, U. An automatic tool for benchmark testing of cloud applications. In: SCITEPRESS. **International Conference on Cloud Computing and Services Science**. [S.l.], 2017. v. 2, p. 729–736. Citado na página 30.

CESAR, B. K. M. **Estudo e extensão da metodologia DAMICORE para tarefas de classificação**. Dissertação (Dissertação de Mestrado) — Instituto de Ciências Matemáticas e de Computação - Universidade de São Paulo, São Carlos, 2016. Citado nas páginas 37, 38, 39 e 53.

Chen, Y.; Winter, S.; Suri, N. Inferring performance bug patterns from developer commits. In: **2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.: s.n.], 2019. p. 70–81. Citado na página 29.

CHERKASOVA, L.; OZONAT, K.; MI, N.; SYMONS, J.; SMIRNI, E. Anomaly? application change? or workload change? In: CITESEER. **the International Conference on Dependable Systems and Networks (DSN'08)**. [S.l.], 2008. p. 452–461. Citado nas páginas 23 e 25.

CILIBRASI, R.; VITÁNYI, P. M. Clustering by compression. **IEEE Transactions on Information theory**, IEEE, v. 51, n. 4, p. 1523–1545, 2005. Citado na página 38.

COHEN, I.; ZHANG, S.; GOLDSZMIDT, M.; SYMONS, J.; KELLY, T.; FOX, A. Capturing, indexing, clustering, and retrieving system history. **ACM SIGOPS Operating Systems Review**, ACM New York, NY, USA, v. 39, n. 5, p. 105–118, 2005. Citado na página 26.

DAEMEN, J.; RIJMEN, V. Reijndael: The advanced encryption standard. **Dr. Dobb's Journal: Software Tools for the Professional Programmer**, Miller Freeman Inc., v. 26, n. 3, p. 137–139, 2001. Citado na página 71.

DAI, T.; DEAN, D.; WANG, P.; GU, X.; LU, S. Hytrace: a hybrid approach to performance bug diagnosis in production cloud infrastructures. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 30, n. 1, p. 107–118, 2018. Citado nas páginas 28 e 30.

DELAMARO, M.; JINO, M.; MALDONADO, J. **Introdução ao teste de software**. [S.l.]: Elsevier Brasil, 2013. Citado nas páginas 21, 22, 33, 34 e 35.

DELGADO-PÉREZ, P.; SÁNCHEZ, A. B.; SEGURA, S.; MEDINA-BULO, I. Performance mutation testing. **Software Testing, Verification and Reliability**, Wiley Online Library, p. e1728, 2020. Citado na página 30.

DEVELOPERS, U. **ppmd - fast archiver program with good compression ratio**. 2020. Disponível em: <https://ubuntu.pkgs.org/14.04/ubuntu-universe-amd64/ppmd_10.1-5_amd64.deb.html>. Citado nas páginas 38 e 44.

FAASSE, S.; BUCEK, J.; SCHMIDT, D. Out of band performance monitoring of server workloads: Leveraging restful api to monitor compute resource utilization and performance related metrics for server performance analysis. In: **Proceedings of the ACM/SPEC International Conference on Performance Engineering**. [S.l.: s.n.], 2020. p. 4–11. Citado nas páginas 28 e 30.

FOURNIER, Q.; EZZATI-JIVAN, N.; ALOISE, D.; DAGENAIS, M. R. Automatic cause detection of performance problems in web applications. In: IEEE. **2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)**. [S.l.], 2019. p. 398–405. Citado nas páginas 22, 28 e 30.

GAILLY, J. loup; ADLER, M. **zlib - A Massively Spiffy Yet Delicately Unobtrusive Compression Library**. 2020. Disponível em: <https://zlib.net/>. Citado na página 44.

GITHUB, I. **GitHub**. 2022. Disponível em: <https://github.com/about>. Citado na página 64.

GLINZ, M. On non-functional requirements. In: IEEE. **15th IEEE International Requirements Engineering Conference (RE 2007)**. [S.l.], 2007. p. 21–26. Citado nas páginas 21 e 34.

GROUP, K. **Vulkan API**: Cross plataform 3d graphics. 2022. Disponível em: <https://www.vulkan.org/>. Citado na página 67.

HUMBATOVA, N.; JAHANGIROVA, G.; BAVOTA, G.; RICCIO, V.; STOCCO, A.; TONELLA, P. Taxonomy of real faults in deep learning systems. In: **Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering**. [S.l.: s.n.], 2020. p. 1110–1121. Citado na página 57.

HUMMER, W.; INZINGER, C.; LEITNER, P.; SATZGER, B.; DUSTDAR, S. Deriving a unified fault taxonomy for event-based systems. In: **Proceedings of the 6th acm international conference on distributed event-based systems**. [S.l.: s.n.], 2012. p. 167–178. Citado na página 56.

JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. **IEEE transactions on software engineering**, IEEE, v. 37, n. 5, p. 649–678, 2011. Citado na página 35.

JIN, G.; SONG, L.; SHI, X.; SCHERPELZ, J.; LU, S. Understanding and detecting real-world performance bugs. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 47, n. 6, p. 77–88, 2012. Citado nas páginas 22 e 26.

JR, H. L. Performance evaluation and monitoring. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 3, n. 3, p. 79–91, 1971. Citado na página 36.

KHATUYA, S.; GANGULY, N.; BASAK, J.; BHARDE, M.; MITRA, B. Adele: Anomaly detection from event log empiricism. In: IEEE. **IEEE INFOCOM 2018-IEEE Conference on Computer Communications**. [S.l.], 2018. p. 2114–2122. Citado na página 30.

KIM, D. **Fluid engine development**. [S.l.]: CRC Press, 2017. Citado nas páginas 66, 67 e 97.

____. **Fluid Engine Dev - Jet**. 2022. Disponível em: <https://github.com/doyubkim/fluid-engine-dev>. Citado nas páginas 66, 97, 98 e 99.

KOCHKOV, D.; SMITH, J. A.; ALIEVA, A.; WANG, Q.; BRENNER, M. P.; HOYER, S. Machine learning–accelerated computational fluid dynamics. **Proceedings of the National Academy of Sciences**, National Acad Sciences, v. 118, n. 21, p. e2101784118, 2021. Citado na página 97.

KOU, H.; CHEN, P. An ensemble signature-based approach for performance diagnosis in big data platform. In: IEEE. **2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)**. [S.l.], 2018. p. 106–115. Citado nas páginas 29 e 30.

LAABER, C.; LEITNER, P. (h| g) opper: Performance history mining and analysis. In: **Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering**. [S.l.: s.n.], 2017. p. 167–168. Citado nas páginas 29 e 30.

LINIETSKY, A. M. J. **Godot Engine**: 2d and 3d cross-platform game engine. 2022. Disponível em: <https://github.com/godotengine/godot>. Citado na página 68.

LIU, M.; ZHANG, Z. Smoothed particle hydrodynamics (sph) for modeling fluid-structure interactions. **Science China Physics, Mechanics & Astronomy**, Springer, v. 62, n. 8, p. 1–38, 2019. Citado na página 98.

LLC, Z. **Zabbix**. 2020. Disponível em: <https://www.zabbix.com/>. Citado na página 37.

MARTINS, L. G.; NOBRE, R.; CARDOSO, J. M.; DELBEM, A. C.; MARQUES, E. Clustering-based selection for the exploration of compiler optimization sequences. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM New York, NY, USA, v. 13, n. 1, p. 1–28, 2016. Citado na página 39.

MAXON. **Cinebench R20**: Evaluate your hardware. 2022. Disponível em: <https://www.maxon.net/en/cinebench>. Citado na página 67.

METSA, J.; KATARA, M.; MIKKONEN, T. Testing non-functional requirements with aspects: An industrial case study. In: IEEE. **Seventh International Conference on Quality Software (QSIC 2007)**. [S.l.], 2007. p. 5–14. Citado na página 35.

MI, N.; CHERKASOVA, L.; OZONAT, K.; SYMONS, J.; SMIRNI, E. Analysis of application performance and its change via representative application signatures. In: IEEE. **NOMS 2008-2008 IEEE Network Operations and Management Symposium**. [S.l.], 2008. p. 216–223. Citado na página 26.

Ming Li; Xin Chen; Xin Li; Bin Ma; Vitanyi, P. M. B. The similarity metric. **IEEE Transactions on Information Theory**, v. 50, n. 12, p. 3250–3264, 2004. Citado na página 44.

MITRA, S.; BRONEVETSKY, G.; JAVAGAL, S.; BAGCHI, S. Dealing with the unknown: Resilience to prediction errors. In: IEEE. **2015 International Conference on Parallel Architecture and Compilation (PACT)**. [S.l.], 2015. p. 331–342. Citado na página 30.

MONGODB, I. **MongoDB**: The mongodb database. 2022. Disponível em: <https://github.com/mongodb/mongo>. Citado na página 68.

MONTES, V. S. **Detecção de defeitos de software utilizando agrupamento de perfis de desempenho**. Dissertação (Dissertação de Mestrado) — Instituto de Ciências Matemáticas e de Computação - Universidade de São Paulo, São Carlos, 2019. Citado nas páginas 23, 24, 25, 41, 42, 43, 45, 46, 47, 48, 49, 50, 51, 52, 54, 57, 68, 69, 70, 71, 72, 73, 80 e 107.

MONTES, V. S. **Network Telemetry Benchmark**: Códigos-fonte dos aplicativos yatserver/yatclient e ferramentas auxiliares. 2019. Disponível em: <https://gitlab.com/vmontes/networktelemetry>. Citado na página 47.

____. **Tricorder Framework**: Conjunto de ferramentas e scripts usados na aplicação do método tricorder. 2019. Disponível em: <https://gitlab.com/vmontes/tricorder>. Citado nas páginas 42, 43 e 50.

MORELL, L. A theory of fault-based testing. **IEEE Transactions on Software Engineering**, v. 16, n. 8, p. 844–857, 1990. Citado na página 35.

MORO, L. F. d. S.; RODRIGUES, C. L.; ANDRADE, F. R.; DELBEM, A. C.; ISOTANI, S. Caracterização de alunos em ambientes de ensino online: Estendendo o uso da damicore para minerar dados educacionais. In: **Anais dos Workshops do Congresso Brasileiro de Informática na Educação**. [S.l.: s.n.], 2014. v. 3, n. 1, p. 631. Citado na página 39.

MOZILLA. **MozJPEG**: Mozilla jpeg encoder project. 2022. Disponível em: <https://github.com/mozilla/mozjpeg>. Citado na página 67.

MÜHLBAUER, S.; APEL, S.; SIEGMUND, N. Accurate modeling of performance histories for evolving software systems. In: IEEE. **2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2019. p. 640–652. Citado nas páginas 29 e 30.

MYERS, G. J.; SANDLER, C. *et al.* **The art of software testing**. [S.l.]: John Wiley & Sons,, 2004. Citado nas páginas 21, 22, 33, 34 e 35.

NEWMAN, M. E. Fast algorithm for detecting community structure in networks. **Physical review E**, APS, v. 69, n. 6, p. 066133, 2004. Citado nas páginas 38 e 39.

NVIDIA. Nvidia nvenc obs guide. **Nvidia Corporation**, 2022. Disponível em: <https://www.nvidia.com/en-us/geforce/guides/broadcasting-guide/>. Citado na página 89.

____. Nvidia system management interface. **Nvidia Corporation**, 2022. Disponível em: <https://developer.nvidia.com/nvidia-system-management-interface>. Citado nas páginas 60 e 89.

OBS, S. C. **Open Broadcaster Software Studio**: Free and open source software for video recording and live streaming. 2022. Disponível em: <https://obsproject.com/>. Citado nas páginas 65, 66 e 88.

OPENSSL. **OpenSSL**: Cryptography and ssl/tls toolkit. 2022. Disponível em: <https://www.openssl.org/>. Citado nas páginas 65 e 71.

POSTGRESQL, G. D. G. **PostgreSQL**: Postgresql database management system. 2022. Disponível em: <https://github.com/postgres/postgres>. Citado na página 68.

PROJECT, G. **Ganglia Monitoring System**. 2020. Disponível em: <http://ganglia.info/>. Citado na página 37.

QT, T. Q. C. **Qt Documentation**. 2020. Disponível em: <https://doc.qt.io/>. Citado na página 47.

REICHELT, D. G.; KÜHNE, S. How to detect performance changes in software history: Performance analysis of software system versions. In: **Companion of the 2018 ACM/SPEC International Conference on Performance Engineering**. [S.l.: s.n.], 2018. p. 183–188. Citado na página 30.

REICHELT, D. G.; KÜHNE, S.; HASSELBRING, W. Peass: a tool for identifying performance changes at code level. In: IEEE. **2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2019. p. 1146–1149. Citado na página 30.

RODOLA, G. **Psutil Documentation**. 2020. Disponível em: <https://psutil.readthedocs.io/en/latest/>. Citado nas páginas 36, 37 e 42.

ROGSTAD, E.; BRIAND, L.; TORKAR, R. Test case selection for black-box regression testing of database applications. **Information and Software Technology**, Elsevier, v. 55, n. 10, p. 1781–1795, 2013. Citado na página 29.

SAITOU, N.; NEI, M. The neighbor-joining method: a new method for reconstructing phylogenetic trees. **Molecular biology and evolution**, v. 4, n. 4, p. 406–425, 1987. Citado na página 38.

Sanches, A.; Cardoso, J. M. P.; Delbem, A. C. B. Identifying merge-beneficial software kernels for hardware implementation. In: **2011 International Conference on Reconfigurable Computing and FPGAs**. [S.l.: s.n.], 2011. p. 74–79. Citado nas páginas 23, 38, 43 e 44.

SÁNCHEZ, A. B.; DELGADO-PÉREZ, P.; MEDINA-BULO, I.; SEGURA, S. Search-based mutation testing to improve performance tests. In: **Proceedings of the Genetic and Evolutionary Computation Conference Companion**. [S.l.: s.n.], 2018. p. 316–317. Citado na página 30.

SÁNCHEZ, A. B.; DELGADO-PÉREZ, P.; SEGURA, S.; MEDINA-BULO, I. Performance mutation testing: Hypothesis and open questions. **Information and Software Technology**, Elsevier, v. 103, p. 159–161, 2018. Citado na página 30.

SAUVANAUD, C.; SILVESTRE, G.; KAÂNICHE, M.; KANOUN, K. Data stream clustering for online anomaly detection in cloud applications. In: IEEE. **2015 11th European Dependable Computing Conference (EDCC)**. [S.l.], 2015. p. 120–131. Citado nas páginas 28 e 30.

SBA, R. **Simcoin**: A blockchain simulation framework. 2022. Disponível em: <https://github.com/sbaresearch/simcoin>. Citado na página 67.

SHAOWDY. **Multithreaded-Algorithms**: Mergesort / parallel mergesort / parallel recursive quicksort / java's parallel sort. 2017. Disponível em: <https://github.com/shaowdy/Multithreaded-Algorithms>. Citado nas páginas 65 e 79.

SHEN, D.; LUO, Q.; POSHYVANYK, D.; GRECHANIK, M. Automating performance bottleneck detection using search-based application profiling. In: **Proceedings of the 2015 International Symposium on Software Testing and Analysis**. [S.l.: s.n.], 2015. p. 270–281. Citado nas páginas 22, 28 e 30.

SILVA, V.; NEVES, L.; SOUZA, R.; COUTINHO, A. L.; OLIVEIRA, D. de; MATTOSO, M. Adding domain data to code profiling tools to debug workflow parallel execution. **Future Generation Computer Systems**, Elsevier, 2018. Citado na página 30.

SOUSA, V. S.; NEVES, L.; SOUZA, R.; COUTINHO, A.; OLIVEIRA, D. de; MATTOSO, M. Integrating domain-data steering with code-profiling tools to debug data-intensive workflows. In: **WORKS@SC**. [S.l.: s.n.], 2016. Citado nas páginas 22, 28 e 30.

SUJON, M.; SHAFIUZZAMAN, M.; RAHMAN, M. M.; RAHMAN, R. Characterization and localization of performance-bugs using naive bayes approach. In: IEEE. **2016 5th International Conference on Informatics, Electronics and Vision (ICIEV)**. [S.l.], 2016. p. 791–796. Citado nas páginas 29 e 30.

TULLY, S. **Crypto-Example**: A short, proof-of-concept rsa and aes encryption program with openssl. 2013. Disponível em: <https://github.com/shanet/Crypto-Example>. Citado nas páginas 65 e 71.

UNIGINE. **Heaven Benchmark**: Extreme performance and stability test for pc hardware. 2022. Disponível em: <https://benchmark.unigine.com/heaven>. Citado na página 67.

VIDEOLAN. **VLC Media Player**: Free multimedia solutions for all os! 2022. Disponível em: <https://www.videolan.org/index.pt.html>. Citado na página 88.

____. **x264 encoder**: Free software library and application for encoding video streams. 2022. Disponível em: <https://www.videolan.org/developers/x264.html>. Citado na página 89.

VOKOLOS, F. I.; WEYUKER, E. J. Performance testing of software systems. In: **Proceedings of the 1st International Workshop on Software and Performance**. [S.l.: s.n.], 1998. p. 80–87. Citado na página 35.

WANG, T.; ZHANG, W.; WEI, J.; ZHONG, H. Fault detection for cloud computing systems with correlation analysis. In: IEEE. **2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)**. [S.l.], 2015. p. 652–658. Citado nas páginas 29 e 30.

WANG, X.; JIN, Y.; YU, Y. A mobile network performance evaluation method based on multivariate time series clustering with auto-encoder. In: **Proceedings of the 2nd International Conference on Telecommunications and Communication Engineering**. [S.l.: s.n.], 2018. p. 33–37. Citado na página 30.

WANG, Y.; WU, Z.; LI, Q.; ZHU, Y. A model of telecommunication network performance anomaly detection based on service features clustering. **IEEE Access**, IEEE, v. 5, p. 17589–17596, 2017. Citado na página 30.

WEN, Z.; DAI, W.; ZOU, D.; JIN, H. Perfdoc: Automatic performance bug diagnosis in production cloud computing infrastructures. In: IEEE. **2016 IEEE Trustcom/BigDataSE/ISPA**. [S.l.], 2016. p. 683–690. Citado nas páginas 28 e 30.

WERT, A.; SCHULZ, H.; HEGER, C. Aim: Adaptable instrumentation and monitoring for automated software performance analysis. In: IEEE. **2015 IEEE/ACM 10th International Workshop on Automation of Software Test**. [S.l.], 2015. p. 38–42. Citado nas páginas 28 e 30.

WIENKE, J.; WREDE, S. Autonomous fault detection for performance bugs in component-based robotic systems. In: IEEE. **2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)**. [S.l.], 2016. p. 3291–3297. Citado na página 30.

Wurzenberger, M.; Skopik, F.; Fiedler, R.; Kastner, W. Applying high-performance bioinformatics tools for outlier detection in log data. In: **2017 3rd IEEE International Conference on Cybernetics (CYBCONF)**. [S.l.: s.n.], 2017. p. 1–8. Citado nas páginas 29 e 30.

ZHU, X.; DAVIDSON, I. **Knowledge discovery and data mining: challenges and realities**. [S.l.]: Information Science Reference Hershey, 2007. Citado na página 37.


ICMC
USP
SÃO CARLOS