DESENVOLVIMENTO DE PROTÓTIPO PARA ANÁLISE DE OBJETOS DINÂMICOS E IDENTIFICAÇÃO DE VAZAMENTOS DE MEMÓRIA EM APLICAÇÕES DELPHI

Henry de Souza dos Santos [1], Marcel Campos Inocencio [2]

Resumo: O presente trabalho aborda o problema de vazamentos de memória em aplicações Delphi, causado pela alocação de objetos dinâmicos sem a devida liberação, o que compromete o desempenho e a estabilidade do software. O objetivo principal foi a automatização da análise e identificação de vazamentos de memória de objetos dinâmicos em aplicações Delphi. A metodologia empregada consistiu em análise estática do código, permitindo a inspeção automática do código-fonte em busca de objetos dinâmicos não liberados. Nos testes realizados em projetos Delphi de diferentes portes, o protótipo identificou os vazamentos de memória com precisão superior a 95% e uma taxa de falsos positivos inferior a 2%. Sua contribuição ocorre de forma prática ao auxiliar desenvolvedores na detecção prévia de problemas de gerenciamento de memória, complementando práticas tradicionais de testes e melhorando a qualidade e a estabilidade das aplicações, sem a necessidade de executar o software.

Palavras-chave: vazamento de memória; análise estática; Delphi; objetos dinâmicos.

ABSTRACT: This paper addresses the issue of memory leaks in Delphi applications, caused by the allocation of dynamic objects without proper release, which affects software performance and stability. The primary objective was to automate the analysis and identification of memory leaks of dynamic objects in Delphi applications. The methodology employed involved static code analysis, enabling automatic inspection of source code to detect unreleased dynamic objects. In tests conducted on Delphi projects of various sizes, the prototype identified memory leaks with an accuracy greater than 95% and a false-positive rate below 2%. Its practical contribution lies in aiding developers in the early detection of memory management issues, complementing traditional testing methods and enhancing application quality and stability without requiring software execution.

[1]Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense (Unesc), henrysantoss29@unesc.net

[2]Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense (Unesc), marcel.inocencio@gmail.com

Keywords: memory leak; static analysis; Delphi; dynamic objects.

1 INTRODUÇÃO

A memória RAM é um componente crítico em sistemas computacionais, atuando como armazenamento temporário de dados e instruções essenciais para a execução de aplicações (Cai et al., 2020). Seu gerenciamento eficiente é vital para garantir desempenho e estabilidade, especialmente em softwares que demandam processamento em tempo real e manipulação de grandes volumes de dados (Patterson; Hennessy, 2020).

No entanto, vazamentos de memória e falhas na liberação de recursos alocados dinamicamente representam um desafio persistente, levando à degradação gradual do desempenho, instabilidade e até falhas críticas em aplicações (Amalfitano et al., 2020). Estudos como os de Aslanyan et al. (2022) e Varjao (2019) destacam que vazamentos ocorrem quando objetos dinâmicos não são desalocados adequadamente, resultando em desperdício de recursos e comprometendo a eficiência operacional.

Em linguagens de programação como Delphi, amplamente utilizada na indústria por sua flexibilidade e velocidade, a ausência de ferramentas nativas para detecção de vazamentos de memória dificulta a manutenção de código robusto (Cantú, 2019). O gerenciamento de memória é parcialmente manual, a responsabilidade pela alocação e liberação de objetos recai sobre o desenvolvedor e erros nesse processo acabam sendo comuns e podem gerar vazamentos insidiosos, difíceis de detectar sem ferramentas especializadas (Jain et al., 2020).

Nesta situação, a gestão eficiente de memória é crucial para a estabilidade e escalabilidade de aplicações, especialmente em sistemas críticos como servidores ou soluções corporativas (Glentis; Angelopoulos, 2019; Alves, 2009).

A análise de objetos dinâmicos surge como uma estratégia promissora para identificar padrões de alocação e liberação, oferecendo ideias para otimizar o uso da memória (Hof, 2020).

Além disso, a adoção de tecnologias como a coleta de lixo e técnicas de prevenção como o uso de estruturas para liberação de memória é essencial para mitigar riscos, destacando a necessidade de soluções automatizadas que auxiliem desenvolvedores a garantir a robustez de suas aplicações (Gabrijelcic, 2019).

Pesquisas anteriores abordaram a detecção de vazamentos de memória em diferentes contextos. Costa (2018) explorou o monitoramento

de envelhecimento de software, utilizando indicadores como RSS (Resident Set Size), representando a porção da memória ocupada por um processo que está fisicamente presente na memória RAM, e HSS (Heap Space Size) indicando a quantidade de memória utilizada para armazenar objetos criados dinamicamente durante a execução de um programa. Conforme Hu et al. (2021), esses indicadores são essenciais para identificar vazamentos em ambientes reais.

Sena (2017) conduziu uma investigação aprofundada sobre vazamentos de memória em condições de uso real, empregando análises estatísticas rigorosas para identificar tendências de degradação associadas a esse fenômeno. Seu estudo comparou o comportamento de diferentes versões de software sob cargas controladas e demonstrou elevada eficácia na detecção de vazamentos reais, distinguindo-os de variações normais do sistema.

Em C/C++, Jain et al. (2020) propuseram a biblioteca MLD (Memory Leak Detection), baseada em grafos, para rastrear objetos não referenciados. Yuan et al. (2022) avançaram com o ZkCheck, uma ferramenta capaz de empregar máquinas de estado e algoritmos de correspondência *fuzzy* (aproximação por similaridade) para detectar padrões críticos no código, especialmente úteis na identificação antecipada de possíveis falhas ou vulnerabilidades devido a vazamentos de memória.

Estas pesquisas, porém, focaram em linguagens como C/C++ ou ambientes genéricos, deixando uma lacuna para soluções específicas em Delphi, onde o gerenciamento da memória exige abordagens customizadas.

Considerando a necessidade de aprimorar o gerenciamento de memória em aplicações Delphi, este trabalho propõe a automatização da análise e identificação de vazamentos de memória de objetos dinâmicos em aplicações Delphi.

Para isso, os objetivos específicos são: fundamentar, por meio de estudo teórico, as causas e padrões de vazamentos em objetos dinâmicos, com ênfase no ambiente Delphi; desenvolver uma ferramenta que identifique práticas inadequadas de alocação de memória; e, por fim, avaliar a eficácia da solução por meio de métricas como precisão na detecção de vazamentos de memória, número de vazamentos identificados corretamente e redução de falsos positivos.

2 MATERIAIS E MÉTODOS

O estudo descreve a criação e a validação de um sistema computacional focado na detecção de vazamentos de memória em projetos desenvolvidos na linguagem Delphi. A iniciativa se caracteriza como uma pesquisa tecnológica de aplicação prática, tendo como principal preocupação a adoção de boas práticas de gerenciamento de recursos e a automação do processo de análise estática.

2.1 AMBIENTE DE DESENVOLVIMENTO E FERRAMENTAS

Para a construção do sistema, utilizou-se um computador executando o sistema operacional Windows 10, equipado com 8 GB de memória RAM e um processador de 3.0 GHz. Tais especificações forneceram a base necessária para analisar múltiplos arquivos (*.pas* e *.dproj*) de tamanhos variados de forma eficiente e sem comprometer o desempenho.

A linguagem Python (versão 3.11) foi escolhida devido à sua flexibilidade no processamento de arquivos de texto, característica essencial quando se deseja identificar padrões específicos em trechos de código.

A biblioteca *Pyparsing*, implementada inteiramente em Python, foi empregada para analisar padrões textuais nos arquivos de código-fonte durante a análise estática conduzida. O *Pyparsing* fornece um conjunto de classes para definição de gramáticas e reconhecimento de padrões sintáticos de forma programática, sem exigir a escrita de uma gramática formal completa da linguagem-alvo.

A biblioteca *Tkinter* foi utilizada para construir a interface gráfica (GUI) da aplicação, permitindo a criação de janelas, botões de ação, barras de progresso e áreas de log de forma intuitiva e consistente.

Além disso, a biblioteca *threading* foi empregada para possibilitar a execução simultânea de tarefas em segundo plano, garantindo que a interface permanecesse responsiva durante o processamento de arquivos e outras operações intensivas. Por fim, integrou-se a biblioteca *webbrowser* à aplicação para abrir automaticamente os relatórios gerados ao término de cada análise, facilitando a visualização imediata dos resultados pelo usuário.

Para organizar todo o desenvolvimento e depuração, recorreu-se ao Visual Studio Code, IDE reconhecida por integrar nativamente com Python e oferecer suporte à ferramenta de controle de versão Git.

Desde o início do projeto, priorizou-se a aplicação de princípios sólidos da Engenharia de Software, particularmente no que diz respeito à

modularidade e manutenibilidade. Diante disso, o código-fonte foi organizado em seções independentes, com objetivos claramente delimitados, facilitando eventuais refatorações ou expansões futuras.

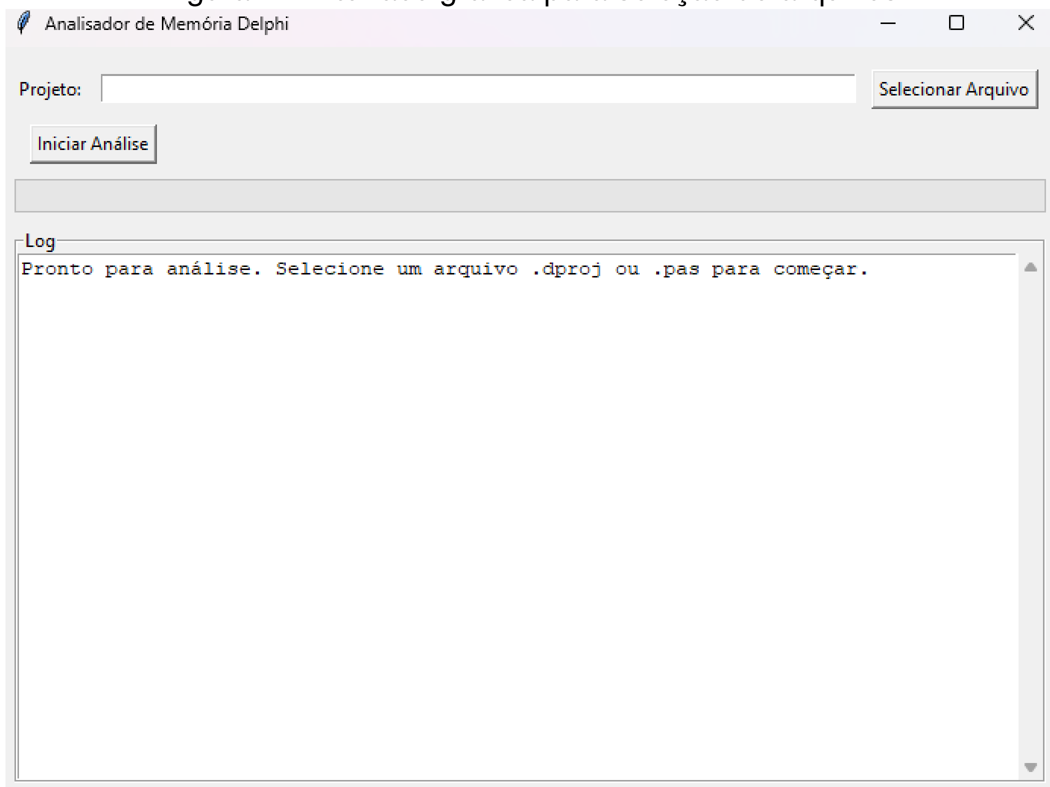
2.2 ESTRUTURA MODULAR DA APLICAÇÃO

Com o intuito de possibilitar uma compreensão clara e uma evolução contínua do sistema, optou-se por dividi-lo em quatro módulos principais, a saber: Interface Gráfica (GUI), Parser de Projetos, Analisador Estático e Gerador de Relatórios.

2.2.1 Interface Gráfica (GUI)

A Interface GUI oferece uma experiência simplificada ao usuário, concentrando todas as funções em uma única janela, conforme a Figura 1. Nesse local, é possível selecionar arquivos *.dproj* ou *.pas* para análise, indicar se deseja um relatório detalhado e acompanhar em tempo real o avanço da inspeção por meio de uma barra de progresso. Ao longo desse processo, o sistema produz mensagens que detalham as etapas em curso, trazendo mais transparência e confiança.

Figura 1 - Interface gráfica para seleção de arquivos.



Fonte: Elaborado pelo autor.

2.2.2 Módulo Parser

Este módulo realiza a leitura do arquivo *.dproj* para identificar todas as unidades *.pas* que constituem um projeto Delphi. Dado que os arquivos *.dproj* seguem um formato XML, o sistema consegue mapear as dependências internas, garantindo que cada parte do projeto seja incluída na verificação. Essa característica é especialmente útil em projetos mais elaborados, que utilizam componentes adicionais ou bibliotecas de terceiros.

2.2.3 Módulo Analisador Estático

O módulo Analisador Estático desempenha um papel fundamental na identificação de potenciais vazamentos de memória em projetos Delphi, sem a necessidade de executar a aplicação. Seu funcionamento consiste na varredura minuciosa dos arquivos fonte (*.pas*), focando especialmente em métodos críticos para a gestão de memória como *Create*, *Free*, *Dispose* e *FreeAndNil*.

Durante a análise estática, são avaliados padrões específicos no código-fonte para identificar inconsistências que possam causar problemas futuros, especialmente relacionados à gestão incorreta de memória e recursos.

Entre os principais padrões aplicados no analisador estão objetos dinâmicos criados e não liberados adequadamente após o uso, referências não inicializadas e erros no emprego de estruturas condicionais ou laços de repetição. De acordo com Silva (2010) e Braga (2023), esses problemas frequentemente causam comportamentos inesperados, falhas críticas ou vazamentos contínuos de memória, levando à degradação do desempenho e da estabilidade do software ao longo do tempo.

Cada ocorrência identificada é registrada detalhadamente pelo sistema, apontando o problema específico, a localização exata no código, facilitando ajustes rápidos e pontuais pelos desenvolvedores.

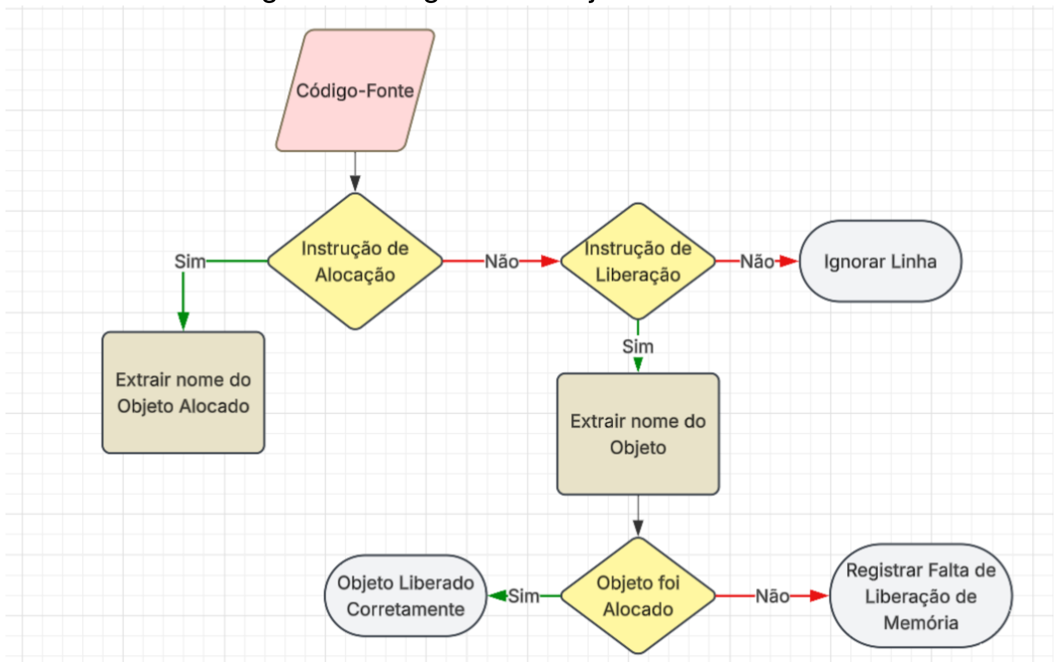
A Figura 2 apresenta um exemplo da análise realizada pelo módulo, ilustrando como o sistema identifica corretamente a criação e a subsequente liberação dos objetos dinâmicos. Destacam-se visualmente as situações em que o objeto não é devidamente liberado, ajudando o usuário a entender rapidamente os problemas detectados.

Figura 2 - Fluxograma Analisador Estático.

Fonte: Elaborado pelo autor.

Para complementar a análise visual, a Figura 3 exibe o detalhamento do relatório preliminar gerado após a varredura do código. Neste relatório inicial, são explicitados claramente os métodos utilizados, permitindo ao usuário a rápida identificação das linhas de código com objetos com possíveis vazamentos de memória.

Figura 3 - Registro de objetos no relatório.

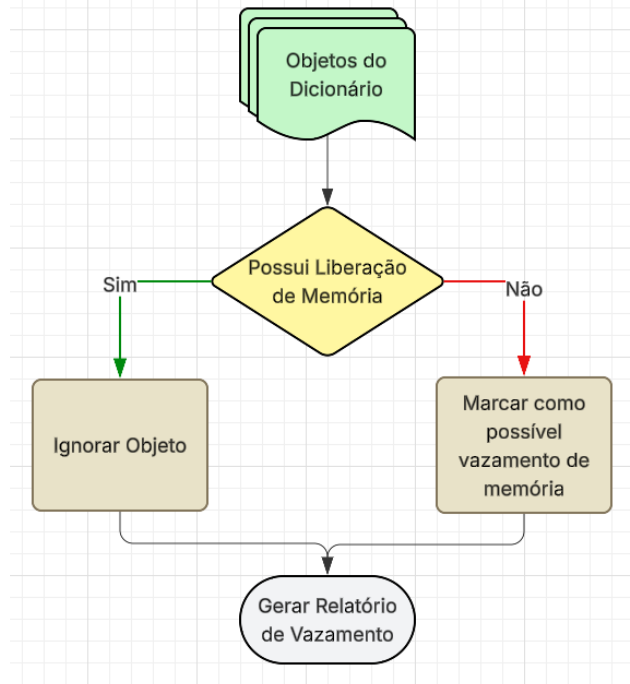


Fonte: Elaborado pelo autor.

Finalmente, na Figura 4 está representado o relatório completo em formato HTML, exibido após a finalização da análise estática. Este

documento contém as linhas exatas onde foram detectados problemas potenciais, descrevendo de maneira clara o tipo de erro e sugerindo ações corretivas para evitar futuros vazamentos.

Figura 4 - Finalização do processo da aplicação.



Fonte: Elaborado pelo autor.

2.2.4 Gerador de Relatórios

O módulo Gerador de Relatórios é responsável por compilar e apresentar, de forma organizada e compreensível, os resultados da análise estática realizada pelo sistema. Após a conclusão da varredura do projeto Delphi, o resultado da inspeção é consolidado em um único arquivo HTML com o estilo CSS incorporado.

Esse formato autônomo facilita a visualização detalhada dos problemas encontrados em qualquer navegador, além de promover uma rápida identificação e correção dos vazamentos de memória detectados.

A Figura 5 apresenta o relatório inicial com um resumo da análise efetuada. Destaca-se o número total de objetos dinâmicos que não foram liberados corretamente e a quantidade de arquivos do projeto afetados por esse problema.

Além disso, o relatório exibe tabelas classificando os erros por frequência, incluindo os tipos de objetos mais frequentemente envolvidos em vazamentos de memória e os arquivos fonte que concentram o maior número de ocorrências de falhas. Esse resumo permite ao desenvolvedor

identificar rapidamente os elementos mais críticos, priorizando as correções necessárias para melhorar a qualidade do código.

Figura 5 - Tabela de erros mais frequentes.

Relatório de Vazamento de Memória: ClockProject.dproj

Gerado em: 28/05/2025 22:13:45

Resumo da Análise

Total de objetos não liberados: **3**

Total de arquivos com problemas: **1**

Objetos ordenados por quantidade de possíveis vazamentos

Tipo	Contagem
TFDQuery	2
TFDMemTable	1

Arquivos com mais objetos não liberados

Arquivo	Objetos
Justify.pas	3

Fonte: Elaborado pelo autor.

Na Figura 6, é possível observar uma visão detalhada das faltas de liberação de memória organizadas por arquivo. Este relatório fornece informações específicas sobre cada objeto com possível vazamento de memória, indicando claramente as linhas do código-fonte onde os objetos foram alocados sem a respectiva liberação adequada. Com essa estrutura, o desenvolvedor pode atuar diretamente nas áreas mais impactadas.

Figura 6 - Arquivos individuais e seus erros encontrados.

Detalhes por Arquivo

Justify.pas 3

Caminho: C:/Faculdade/clock-main/api/Justify.pas

InsertJustify (procedure) - Linha 28

Nome do Objeto	Tipo	Linha no Arquivo	Declaração
loQry	TFDQuery	32	`loQry (TFDQuery)`

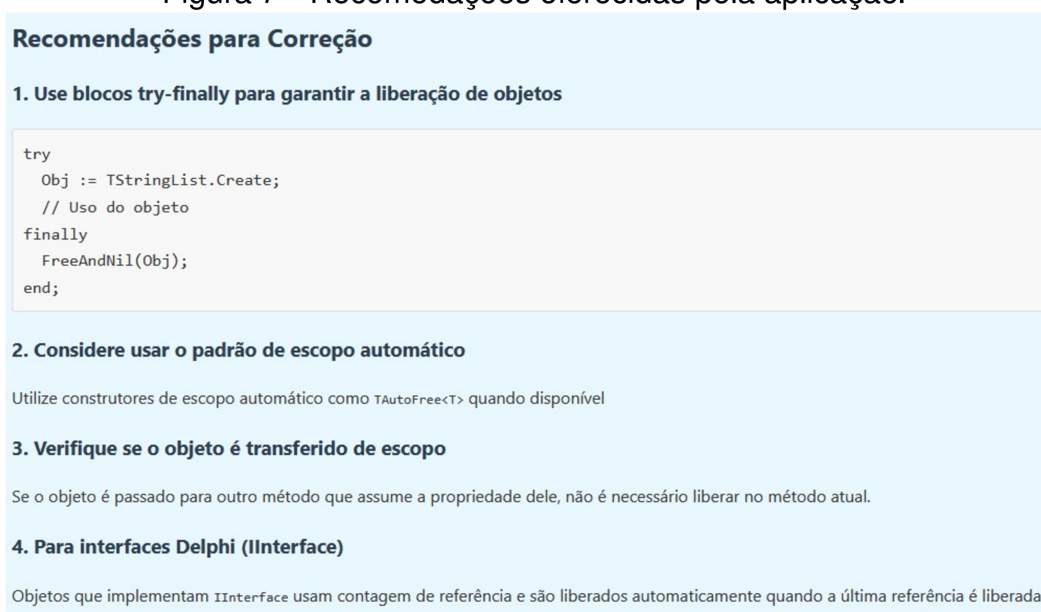
AcceptJustify (procedure) - Linha 106

Nome do Objeto	Tipo	Linha no Arquivo	Declaração
loDados	TFDMemTable	108	`loDados (TFDMemTable)`
loQryDelay	TFDQuery	110	`loQryDelay (TFDQuery)`

Fonte: Elaborado pelo autor.

Por fim, a Figura 7 exemplifica uma seção do relatório que sugere métodos e práticas recomendadas para resolver as falhas detectadas. Este documento não apenas aponta os erros, bem como oferece orientações concretas e boas práticas de programação em Delphi, como a utilização apropriada dos métodos *Free* e *FreeAndNil*, além do correto emprego das estruturas *try-finally*, colaborando assim para uma gestão mais segura e eficiente dos recursos dinâmicos alocados.

Figura 7 - Recomedações oferecidas pela aplicação.



Fonte: Elaborado pelo autor.

2.3 FUNCIONAMENTO GERAL E FLUXO DE USO

Para utilizar o sistema, o usuário abre a interface principal, seleciona o arquivo *.dproj* do projeto ou o arquivo *.pas* desejado e aciona o botão "Iniciar Análise". O processamento ocorre em segundo plano, evitando a interrupção da interação. A barra de progresso indica o quão avançada está a varredura, e o log textual aponta, a cada instante, o status das operações. Ao final, apresenta-se um link que leva ao relatório HTML, contendo uma lista detalhada dos trechos de código que precisam de atenção.

2.4 AMBIENTE DE TESTES E VALIDAÇÃO

Para avaliar a consistência e a eficácia da ferramenta, foram elaborados testes com diferentes perfis de projetos:

1. Exemplos Controlados: pequenos trechos de código projetados propositalmente para apresentar falhas de liberação de objetos, tais como

a ausência de *FreeAndNil* ou a não utilização da estrutura *try-finally*. Esses exemplos serviram para verificar se a aplicação apontava corretamente e com precisão os problemas analisados;

2. Projetos Públicos Reais de Pequeno e Médio Porte: foram utilizados códigos-fonte disponíveis publicamente na internet, obtidos em repositórios abertos e projetos comunitários, contendo formulários visuais e múltiplas referências externas;

Em cada um desses cenários, a ferramenta apontou as linhas exatas do código que poderiam causar vazamentos de memória. Após a correção dos trechos indicados, realizaram-se testes adicionais que comprovaram a ausência de novos vazamentos nos pontos anteriormente citados.

2.5 PERSPECTIVAS DE USO E RELEVÂNCIA

O sistema descrito demonstra potencial para mitigar riscos de vazamentos de memória em projetos Delphi, contribuindo para um ciclo de desenvolvimento mais seguro e econômico. Sua arquitetura modular possibilita sua incorporação a processos de Integração Contínua (CI) ou a ferramentas de gestão de qualidade, de modo que cada nova alteração no código possa ser analisada antes que a mesma seja colocada em produção.

Dessa forma, a ferramenta pode se tornar um complemento valioso tanto para projetos em fase inicial de desenvolvimento quanto para aplicações legadas que carecem de análises contínuas.

3 DISCUSSÃO E RESULTADOS

Os testes realizados com o protótipo desenvolvido evidenciaram sua eficácia na detecção de vazamentos de memória em aplicações Delphi de diferentes portes. Assim, foram avaliados três níveis de complexidade em projetos de software: simples, intermediários e complexos.

- Projetos simples: caracterizados por apresentar até 5 arquivos-fonte, estruturas de código básicas e nenhuma dependência externa;
- Projetos intermediários: envolveram uma quantidade de até 15 arquivos-fonte, com média de 120 objetos, tais como classes ou componentes do software. Possuíam formulários visuais, interfaces gráficas e referenciavam até 5 bibliotecas externas;

- Projetos complexos: possuíam como estrutura um grande volume de até 50 arquivos-fonte e em média 300 objetos, fazendo uso de no máximo 15 bibliotecas externas e empregando formulários visuais e interfaces gráficas;

Os resultados obtidos em 4 testes para cada um desses níveis de complexidade são apresentados na Tabela 1.

Tabela 1 – Métricas de detecção de vazamentos de memória por nível de complexidade do projeto.

Complexidade do projeto	Arquivos analisados	Objetos analisados	Vazamentos existentes	Vazamentos detectados
Simples	5	30	3	3
Intermediário	15	120	12	12
Complexo	50	300	28	25

Fonte: Elaborado pelo autor.

Em complemento a essas métricas gerais, a Tabela 2 detalha informações essenciais relacionadas ao desempenho e à qualidade da análise. Nela estão apresentados os resultados obtidos quanto à precisão percentual na detecção dos vazamentos, o tempo médio decorrido para realizar as análises e a quantidade de falsos positivos observados em cada nível de complexidade avaliado. Essas informações permitem avaliar, de maneira objetiva, o desempenho prático da ferramenta, especialmente em contextos de crescente complexidade.

Tabela 2 – Precisão, desempenho e número de falsos positivos por nível de complexidade dos projetos analisados.

Complexidade do projeto	Precisão (%)	Tempo de análise (s)	Falsos positivos
Simples	100	0,7	0
Intermediário	85,7	2,9	2
Complexo	73,5	8,4	9

Fonte: Elaborado pelo autor.

A avaliação do desempenho foi realizada utilizando a métrica de precisão com base no percentual de vazamentos detectados corretamente em comparação com o total de vazamentos reais em cada projeto considerado, corrigida pela frequência de falsos positivos na análise.

A análise apresentou um desempenho destacado da ferramenta com taxas de precisão diminuindo à medida que a complexidade dos pro-

jetos aumentou, atingindo 100% de precisão para projetos simples, onde foram examinados em média 30 objetos, dos quais 3 tiveram vazamentos. Nos projetos de nível intermediário, atingiu 85,7% de precisão em cerca de 120 objetos, com 12 vazamentos identificados. E nos projetos complexos, houve 73,5% de precisão, nos quais foram avaliados em média 300 objetos, com 25 vazamentos de memória identificados.

No pior caso analisado, a taxa de acerto permaneceu acima de 70%, sendo considerada elevada, o que demonstra a robustez da ferramenta. Quanto ao desempenho, observou-se que o tempo de execução da análise aumentou à medida que crescia o número de objetos verificados.

Além disso, a varredura das unidades de código ocorreu de forma suficientemente rápida para permitir a integração a fluxos de trabalho, tais como inspeções frequentes ou processos de integração contínua.

Esse resultado indica que a ferramenta é viável para uso contínuo durante o desenvolvimento, mesmo em projetos reais. A varredura das unidades de código ocorreu de forma suficientemente rápida para integrar-se a fluxos de trabalho, como inspeções frequentes ou processos de Integração Contínua.

Ao examinar criticamente os resultados, emergiram pontos importantes sobre o comportamento e as premissas da ferramenta. Um achado de destaque foi a relevância do uso correto de estruturas *try-finally* no código.

Os testes revelaram que, em cenários onde essa estrutura de tratamento de exceções não foi aplicada de forma adequada, ocorreram vazamentos de objetos quando exceções eram lançadas antes da chamada de liberação, conforme a Figura 8.

Figura 8 - Rotina sem estrutura *try-finally*.

```
procedure Func;
var
  MyList: TStringList;
begin
  // Criação do objeto
  MyList := TStringList.Create;

  // Simulação de operações que podem lançar uma exceção
  MyList.Add('Item 1');
  if Random < 0.5 then
    raise Exception.Create('Erro simulado durante o processamento');

  // Liberação do objeto - este código não será executado se a exceção for levantada
  MyList.Free;
end;
```

Fonte: Elaborado pelo autor.

Em contrapartida, na Figura 9, em que a estrutura *try-finally* foi utilizada corretamente para garantir a liberação dos objetos, a ferramenta não identificou problemas, confirmando que essa prática protege contra vazamentos mesmo na presença de falhas em tempo de execução.

Esse resultado reforça as boas práticas de gerenciamento manual de memória em Delphi e valida a estratégia do analisador de código em verificar explicitamente a existência (ou ausência) desses padrões.

Figura 9 - Rotina com estrutura *try-finally*.

```
procedure Func;
var
  MyList: TStringList;
begin
  MyList := nil;
  try
    // Criação do objeto
    MyList := TStringList.Create;

    // Simulação de operações que podem lançar uma exceção
    MyList.Add('Item 1');
    if Random < 0.5 then
      raise Exception.Create('Erro simulado durante o processamento');

  finally
    // Liberação garantida do objeto, mesmo se ocorrer exceção
    MyList.Free;
  end;
end;
```

Fonte: Elaborado pelo autor.

Por outro lado, evidencia também uma possível limitação: a ferramenta baseia-se em convenções específicas de código (como o padrão *try-finally*) para inferir vazamentos. Em projetos que adotam estilos diferentes de gerenciamento de recursos, sem seguir essa convenção, a eficácia da detecção pode ser reduzida.

Outro ponto crítico identificado nos experimentos foi a dificuldade de análise envolvendo código compilado ou bibliotecas nativas do Delphi. Em certas situações de teste, o protótipo tentou examinar referências a objetos definidos dentro de *units* internas da biblioteca padrão, o que gerou falsos positivos e dificuldade em rastrear os objetos de forma confiável.

Essas bibliotecas muitas vezes são fornecidas já compiladas ou envolvem implementações internas complexas, fugindo do escopo da análise estática do código-fonte da aplicação. Desta forma, o protótipo mostrou-se limitado quando confrontado com componentes cujo código não estava acessível ou seguia comportamentos internos não previstos nas regras de

detecção.

Um desafio relacionado, observado nas primeiras versões do protótipo, foi a identificação de vazamentos em cenários nos quais um objeto era criado em uma função e liberado somente em outra, ou mantido em uma estrutura de dados global. Inicialmente, a análise estava restrita ao escopo local da função, resultando em dificuldade para correlacionar alocações e liberações que ocorrem em contextos distintos. Após ajustes, o protótipo passou a rastrear melhor esses casos entre procedimentos, mitigando parte do problema.

Entretanto, essa evolução ressalta que a abordagem atual ainda opera principalmente em nível de função/método de forma isolada, sem manutenção de estado persistente entre execuções ou visão global completa do ciclo de vida dos objetos. Logo, vazamentos que se manifestam apenas em execuções longas ou por acumulação de objetos globais podem não ser prontamente identificados na configuração presente. Apesar dessas restrições, a ferramenta apresentou benefícios práticos evidentes.

No trabalho Monitoramento do Envelhecimento de Software Relacionado à Memória: Um Estudo Exploratório de Costa (2018), foram exploradas estratégias de monitoramento do envelhecimento de software por meio de indicadores de consumo de memória em tempo de execução, como RSS (Resident Set Size) e VIRT (Virtual Memory Size).

A ferramenta proposta neste artigo difere fundamentalmente na metodologia: em vez de depender da observação do comportamento do software em execução, adotou-se uma análise estática do código-fonte, eliminando a necessidade de executar o aplicativo para diagnosticar problemas.

Essa distinção metodológica confere à nossa abordagem um caráter preventivo e imediato, possibilitando localizar pontos de vazamento diretamente nas linhas de código antes mesmo da implantação ou do surgimento de sintomas em produção.

Em contrapartida, a técnica de Costa (2018) atua de forma reativa, detectando o envelhecimento à medida que o software roda, e requer um período de observação para coletar dados suficientes (como crescimento sustentado de memória) a fim de disparar alertas, conforme mostrado na Tabela 3. Embora a monitoração em tempo de execução permita capturar o efeito real de um vazamento durante a operação, ela pode sofrer com diagnósticos tardios ou enganosos, caso use indicadores genéricos isoladamente.

Tabela 3 – Comparativo entre o protótipo desenvolvido e a abordagem proposta por Costa (2018).

Aspecto	Protótipo	Costa (2018)
Tipo de detecção	Análise estática do código-fonte	Monitoramento de memória em execução
Ambiente	Desenvolvimento e testes locais	Execução contínua ou produção
Tempo de análise	Imediato, sem execução real	Demorado, requer tempo de execução
Facilidade de uso	Interface com relatórios automáticos	Análise técnica sem interface amigável

Fonte: Elaborado pelo autor.

Costa (2018) demonstra que observar somente a memória física (RSS) pode mascarar vazamentos se a memória virtual continuar crescendo silenciosamente. A solução apresentada contorna esse tipo de armadilha ao inspecionar diretamente padrões de código associados a vazamentos, reduzindo a dependência de inferências indiretas.

Logo, enquanto Costa (2018) foca em indicadores quantitativos de consumo de recursos para inferir o envelhecimento de aplicações em execução, a abordagem deste artigo foca na qualidade do código em si para prevenir que tais degradações ocorram, oferecendo uma solução mais ágil para uso durante o desenvolvimento.

Em relação ao estudo Avaliação da Validade Externa da Técnica de Análise Diferencial para Detecção de Envelhecimento de Software: Um Estudo Confirmatório com Replicação de Sena (2017), que investigou vazamentos de memória sob cargas de trabalho reais, há diferenças significativas tanto de escopo quanto de metodologia. Sena (2017) emprega técnicas estatísticas avançadas, como o filtro de Hodrick-Prescott (HP) combinado a gráficos de controle (CEP) para analisar séries temporais de métricas de memória e identificar tendências de vazamento em aplicações reais.

Essa abordagem de análise diferencial compara o comportamento de duas versões do software executando cenários de carga planejados, gerando gráficos de divergência que destacam anomalias.

Os resultados indicaram alta efetividade: nenhuma ocorrência de falso-negativo e pouquíssimos falso-positivos na detecção dos vazamentos, especialmente quando utilizado o indicador de uso de Heap (HUS) em vez do RSS tradicional.

Ou seja, a técnica de Sena (2017) é bastante confiável em ambiente controlado, conseguindo distinguir vazamentos reais de flutuações normais do sistema. No entanto, essa confiabilidade vem ao custo de uma complexidade experimental elevada, sendo necessário executar a aplicação por horas sob diferentes perfis de carga, coletando milhares de amostras de métricas e aplicando algoritmos de análise estatística para, então, concluir sobre a existência de vazamentos.

Em contraste, o protótipo discutido opera de forma substancialmente mais direta e rápida: não requer execução prolongada nem cenários de teste elaborados, inspecionando o código-fonte em busca de padrões propensos a vazamento.

Enquanto a técnica de Sena (2017) se mostra valiosa para validar e entender o comportamento de sistemas completos em situações de uso real (sendo ideal para análises de qualidade externa do software em estágio final), o protótipo desenvolvido garante melhor a qualidade interna durante a construção do software conforme Tabela 4.

Tabela 4 – Comparativo entre o protótipo desenvolvido e a abordagem proposta por Sena (2017).

Aspecto	**Protótipo**	**Sena (2017)**
Tipo de detecção	Análise estática com foco em vazamentos de objetos	Estatísticas sobre séries temporais (RSS e HUS)
Dados usados	Código-fonte Delphi	Coleta experimental com DTW
Esforço técnico	Baixo, automatizado	Alto, com replicações e pós-processamento
Objetivo	Prevenção de falhas em tempo de codificação	Análise profunda de falhas em execução longa

Fonte: Elaborado pelo autor.

Ressaltando também que o protótipo desenvolvido inclui facilidades práticas, como a geração de relatórios detalhados com interface intuitiva, algo não enfatizado nos trabalhos anteriores.

Essa atenção à usabilidade e apresentação dos resultados melhora a compreensão e a adoção da ferramenta pelos programadores, complementando as contribuições metodológicas com um viés mais aplicado.

Embora metodologicamente distintas, todas as abordagens, tanto

a de Costa (2018) quanto a de Sena (2017) e a presente, compartilham o objetivo de mitigar o problema persistente do vazamento de memória, convergindo na importância de detectar e tratar esse fenômeno para aumentar a confiabilidade e a longevidade dos sistemas de software.

Apesar dos resultados promissores, foram identificadas limitações importantes que abrem oportunidades para melhorias em trabalhos futuros.

Uma das restrições observadas reside na dependência de práticas de codificação padronizadas. Conforme discutido, a eficácia da ferramenta é alta quando o código segue convenções como o uso de estruturas *try-finally* e chamadas explícitas de destrutores (*Dispose*, *Free*, *FreeAndNil*).

4 CONCLUSÃO

Este trabalho teve como objetivo principal desenvolver um protótipo capaz de analisar objetos dinâmicos e identificar vazamentos de memória em aplicações Delphi.

Para isso, adotou-se uma abordagem de análise estática do código-fonte, permitindo ao protótipo inspecionar automaticamente o código em busca de padrões de alocação e liberação que indiquem possíveis vazamentos.

Os experimentos realizados com projetos Delphi de diferentes portes e níveis de complexidade evidenciaram a eficácia do protótipo na detecção de vazamentos de memória.

Nos cenários analisados, o protótipo identificou corretamente os vazamentos, com baixa taxa de falsos positivos. O tempo de análise manteve-se consistente mesmo em casos de maior porte, o que evidencia a viabilidade da solução para uso contínuo durante o desenvolvimento.

Dessa forma, a principal contribuição deste trabalho é oferecer à comunidade de desenvolvedores Delphi uma ferramenta preventiva que auxilia na garantia da qualidade do código. Essa ferramenta é capaz de localizar vazamentos de memória diretamente no código-fonte, antes mesmo da execução da aplicação.

A abordagem estática proposta permite identificar problemas de gerenciamento de memória de forma imediata, complementando práticas tradicionais de testes e depuração e contribuindo para a maior estabilidade e desempenho das aplicações.

Entretanto, algumas limitações foram observadas. A aplicação

mostrou-se restrita ao escopo da análise estática. Em particular, ao lidar com bibliotecas pré-compiladas ou quando a alocação e liberação ocorrem em módulos distintos, a ferramenta pode gerar falsos positivos ou deixar de identificar certos vazamentos. Esses desafios indicam oportunidades de aprimoramento.

Como trabalhos futuros, sugere-se estender o escopo da análise para abranger o ciclo de vida completo dos objetos e integrar técnicas complementares, como a análise dinâmica em tempo de execução. Recomenda-se também realizar testes em larga escala, a fim de validar e aprimorar a robustez da solução.

REFERÊNCIAS

ALVES, M. A. Z. ***Avaliação do Compartilhamento das Memórias Cache no Desempenho de Arquiteturas Multi-Core***. Porto Alegre, Brasil: Universidade Federal do Rio Grande do Sul, 2009.

AMALFITANO, D. et al. ***Do Memories Haunt You? An Automated Black Box Testing Approach for Detecting Memory Leaks in Android Apps***. [S.l.]: IEEE Access, 2020.

ASLANYAN, H. et al. ***Static Analysis Methods For Memory Leak Detection: A Survey***. New York: IEEE, 2022. 1–6 p.

BRAGA, G. A. ***Técnicas de Tolerância a Falhas Controladas por Software para a Proteção do Pipeline de Processadores Gráficos***. Porto Alegre, Brasil: Universidade Federal do Rio Grande do Sul, 2023.

CAI, W. et al. ***Understanding and Optimizing Persistent Memory Allocation***. New York, NY, USA: Association for Computing Machinery, 2020. 60–73 p. Disponível em: <https://doi.org/10.1145/3381898.3397212>. Acesso em: 13 de jan. de 2025.

CANTú, M. ***Desenvolvimento de Software para Windows com Delphi***. [S.l.]: Pearson Education, 2019. ISBN 978-857-609-430-3.

COSTA, B. E. ***Monitoramento do Envelhecimento de Software Relacionado à Memória: Um Estudo Exploratório***. [S.l.]: Universidade Federal de Uberlândia, 2018. Disponível em: <https://repositorio.ufu.br/handle/123456789/24294>. Acesso em: 23 de set. de 2024.

GABRIJELCIC, P. ***Mastering Delphi Programming: A Complete Reference Guide***. [S.l.]: Packt Publishing, 2019. Disponível em: <https://api.semanticscholar.org/CorpusID:214002176>. Acesso em: 4 de fev. de 2025.

GLENTIS, G. O.; ANGELOPOULOS, K. **Leakage detection using leak noise correlation techniques: overview and implementation**

aspects. In: *Proceedings of the 23rd Pan-Hellenic Conference on Informatics*. New York, NY, USA: Association for Computing Machinery, 2019. (PCI '19), p. 50–57. ISBN 9781450372923. Disponível em: <https://doi.org/10.1145/3368640.3368646>. Acesso em: 29 de out. de 2024.

HOF, M. ***Dynamic Object Analysis and Memory Management***. [S.l.]: Springer, 2020.

HU, X. S. et al. ***In-Memory Computing with Associative Memories: A Cross-Layer Perspective***. [S.l.]: 2021 IEEE International Electron Devices Meeting (IEDM), 2021. 25.2.1–25.2.4 p.

JAIN, R. et al. ***Detection of Memory Leaks in C/C++***. [S.l.]: 2020 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS), 2020. 1–6 p.

PATTERSON, D. A.; HENNESSY, J. L. ***Computer Organization and Design MIPS Edition: The Hardware/Software Interface***. [S.l.]: Morgan Kaufmann, 2020.

SENA, G. O. d. ***Avaliação da Validade Externa da Técnica de Análise Diferencial para Detecção de Envelhecimento de Software: Um Estudo Confirmatório com Replicação***. [S.l.]: Universidade Federal de Uberlândia, 2017. Disponível em: <https://repositorio.ufu.br/handle/123456789/19851>. Acesso em: 9 de out. de 2024.

SILVA, A. G. da. ***Utilização de callback no aprimoramento da interface com o usuário***. Porto Alegre, RS, Brasil: Universidade Federal do Rio Grande do Sul, Instituto de Informática, 2010. Disponível em: <https://lume.ufrgs.br/handle/10183/28335>. Acesso em: 10 de mar. de 2025.

VARJAO, F. R. G. ***Gerenciamento dinâmico de memória baseado em regiões com contagem de referências cíclicas***. [S.l.]: Universidade Federal de Pernambuco, 2019. 14–18 p.

YUAN, L. et al. ***MLD: An Intelligent Memory Leak Detection Scheme Based on Defect Modes in Software***. [S.l.]: Entropy, 2022. v. 24. 947 p. Disponível em: <https://www.mdpi.com/1099-4300/24/7/947>. Acesso em: 27 de set. de 2024.