Patrícia Soraia Veríssimo Monteiro

Licenciada em Ciência e Engenharia Informática

Uma Análise Comparativa de Ferramentas de Análise Estática para deteção de Erros de Memória

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: António Maria Lobo César Alarcão Ravara,
Professor Auxilar,
Universidade Nova de Lisboa

Co-orientador: João Manuel dos Santos Lourenço,
Professor Auxiliar,
Universidade Nova de Lisboa

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2018

Uma Análise Comparativa de Ferramentas de Análise Estática para deteção de Erros de Memória

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

A todos aqueles que disseram:
"Tu és capaz."

Agradecimentos

Começo por agradecer ao meu orientador Professor António Ravara e ao meu co-orientador Professor João Lourenço por toda a disponibilidade, sugestões, críticas,"puxões de orelha" e, principalmente, pela vossa paciência. Sem dúvida que todo este percurso contribuiu, não só para a conclusão desta dissertação, mas também para o meu crescimento enquanto futura profissional.

À Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa e em especial ao Departamento de Informática que durante os últimos 6 anos foi a minha segunda casa.

À Débora Viegas que me acompanhou sempre nesta grande aventura chamada Engenharia Informática.

Ao João Tiago, por me obrigar a descansar.

Ao meu namorado, Carlos, pelo apoio, confiança, por me fazer acreditar em mim e por me fazer sorrir, mesmo quando a motivação acabava.

Por fim, quero agradecer aos meus pais, Lélia e Luís, e ao meu irmão, João, por todo o apoio, paciência, constante preocupação e por fazerem com que tudo isto fosse possível.

O trabalho descrito nesta dissertação foi parcialmente enquadrado pelo projeto HiPSTr, refª PTDC/CCI-COM/32456/2017 e LISBOA-01-0145-FEDER-032456.

The sky is the limit.

Resumo

A indústria do software tem evoluído no sentido do desenvolvimento de produtos cada vez mais complexos, no menor tempo possível e com menores custos. As falhas de software estão com frequência associadas a acidentes com graves consequências económicas e/ou humanas, pelo que se torna essencial investir na validação do software, nomeadamente daquele que é crítico. Por este motivo, o software deve ser fortemente testado para evitar erros. Esta dissertação endereça a temática da qualidade do software através de uma análise comparativa da usabilidade e eficácia de quatro ferramentas de análise estática de programas em C/C++. Este estudo permitiu compreender o grande potencial e o elevado impacto que as ferramentas de análise estática podem ter na validação e verificação de software. Como resultado complementar, foram identificados novos erros em programas de código aberto e com elevada popularidade, que foram devidamente reportados.

Palavras-chave: Ferramentas, Estudo comparativo, Análise estática, Erros de software, Validação, Verificação

Abstract

The software industry has evolved towards the development of increasingly complex products, in the shortest possible time and at a lower cost. Software failures are often associated with accidents with serious economic and / or human consequences, so it becomes imperative to invest in software validation, particularly of critical software. For this reason, the software must be heavily tested to avoid errors. This dissertation addresses the thematic of software quality through a comparative analysis of the usability and effectiveness of four static program analysis tools in C / C ++. This study allowed us to understand the great potential and high impact that static analysis tools can have on software validation and verification. As a complementary result, new bugs were identified in open source and high-popularity programs, which were duly reported.

Keywords: Tools, Comparative study, Static analysis, Software errors, Validation, Verification

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Capítulo 1

Introdução

Neste capítulo é apresentada uma breve introdução ao trabalho desenvolvido nesta dissertação, começando pela contextualização do tema (Secção 1.1), identificação do problema (Secção 1.2), objetivo (Secção 1.3) e principais contribuições (Secção 1.4). Por fim, é descrita a estrutura do presente documento (Secção 1.5).

1.1 Contexto

Entre os objetivos das empresas da área da tecnologia de informação consta, naturalmente, o de construir código de qualidade. No entanto, a elevada concorrência no mercado conduz a uma crescente necessidade de desenvolvimento de software cada vez mais complexo, no menor tempo possível e com menores custos. Esta tendência induz, naturalmente, ao aumento da quantidade de erros no software. Intuitivamente, esperamos que um projeto duas vezes maior tenha duas vezes mais erros. No entanto, tal não é verdade, pois a densidade de erros por mil linhas de código também aumenta com o aumento do tamanho do projeto. Projetos muito grandes têm uma densidade de erros até 4 vezes maior que projetos pequenos [44].

Tabela 1.1: Relação entre a dimensão de um projeto e a sua densidade de erros.
(Adaptada de "Code complete" (McConnell, 2004) [44])

Dimensão do projeto (Número de linhas de código)	**Densidade típica de erros** (Por mil linhas de código)
< 2K	0 – 25
2K – 16K	0 – 40
16K – 64K	0.5 – 50
64K – 512K	2 – 70
> 512K	4 – 100

Na Tabela 1.1 está representada a relação entre a dimensão de um projeto e a sua densidade de erros. Os dados desta tabela são referentes a projetos específicos e podem, portanto, diferir de projeto para projeto. Ainda assim, estes dados servem para ilustrar que o número de erros aumenta de forma drástica à medida que o tamanho do projeto aumenta.

Além da relação entre a dimensão do projeto e a densidade de erros identificados no mesmo, quanto mais tarde um problema for detetado, mais difícil e mais caro será resolver esse problema. Na Figura 1.1 está representada a relação entre os custos da correção de erros e a fase do desenvolvimento em que esses erros são descobertos.

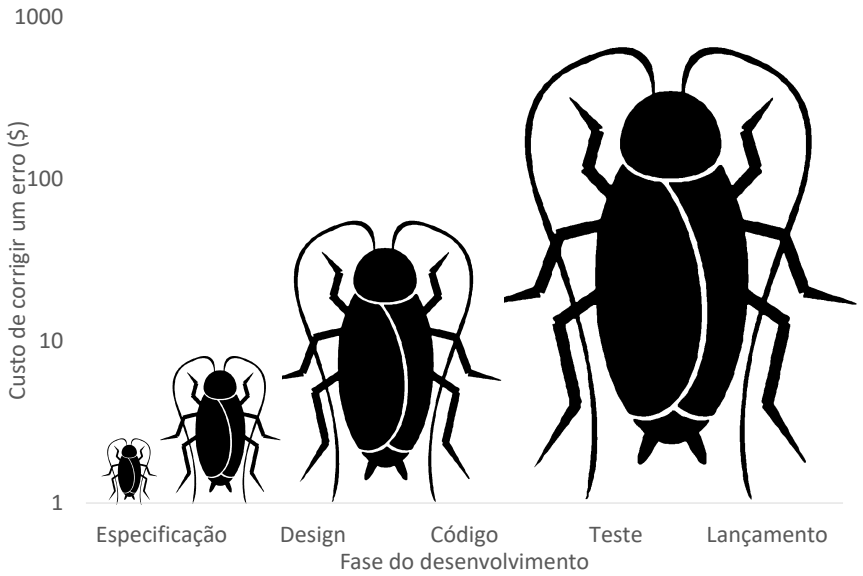


Figura 1.1: Relação entre os custos de corrigir um erro em função da fase em que esse erro é descoberto.

(Figura adaptada de "Software Testing" (R. Patton, 2005) [52])

Os custos são exponenciais, ou seja, aumentam mais do que linearmente com o aumento do tempo. Considerando o exemplo da Figura 1.1, os custos aumentam dez vezes em cada nova fase do desenvolvimento do software. Identificar e corrigir um erro durante a fase inicial do desenvolvimento, quando a especificação está a ser a escrita, tem um custo relativo muito baixo. O mesmo erro, se for encontrado na fase de *design* do software, pode custar dez vezes mais do que na fase anterior. Em último caso, se o erro for detetado pelo cliente, o custo de corrigir este erro será mil vezes maior do que na face inicial [52]. Assim sendo, existe a necessidade de utilizar técnicas que identifiquem problemas no software numa fase tão inicial do seu desenvolvimento quanto possível.

A presença de defeitos no software pode conduzir a problemas variados, tais como erros funcionais (o programa não cumpre os requisitos), falhas e/ou vulnerabilidades de segurança, baixa performance ou a interrupção da execução do programa. Qualquer falha de software deve ser evitada, no entanto, no caso particular do desenvolvimento de *software crítico* é essencial garantir que não existem erros no código. As falhas de *software*

crítico são muitas vezes associadas a desastres com graves consequências económicas e/ou humanas. Por este motivo, os *softwares críticos* exigem níveis muito elevados de cobertura de código, ou seja, exigem testes que executem todas as declarações e condições existentes no código e que sejam revalidados sempre que uma nova funcionalidade é adicionada.

Casos como o erro do Pentium 5 (1994), a autodestruição do Ariane 5 (1996) e do Mars Climate Orbiter (1999), o bloqueio do terminal 5 no Aeroporto de Londres-Heathrow (2008) e, mais recentemente, os ataques Meltdown/Spectre (2017) são exemplos das consequências das falhas de software. Neste último caso, relevado em Dezembro de 2017, um atacante pode tirar vantagem da forma como os processadores Intel, AMD e ARM são construídos atualmente. Essencialmente, o que os processadores fazem é tentar prever as próximas instruções, através de execução especulativa, de forma a otimizar o seu desempenho. Os resultados dessa execução são guardados em cache, permitindo ao processador economizar tempo, caso a sua previsão esteja correta. Se a previsão estiver incorreta, o processador desfaz o código especulativo e executa o código correto. Assim, um atacante pode utilizar os processadores para executar código que acede ilegalmente à memória do computador para depois, no período de tempo entre o acesso à memória e o código executado ser esquecido pelo processador, carregar um único byte em cache. Esta operação é considerada legal, uma vez que o byte carregado pertence à própria memória do programa. Através do tempo de leitura dessa memória, o atacante pode perceber qual é o byte que está em cache. Um byte que não esteja em cache vai demorar mais tempo a ser lido. Sabendo qual é o byte em cache o atacante consegue determinar qual foi o conteúdo da memória ilegal acedida anteriormente pelo processador durante a execução especulativa. Ao repetir estas etapas várias vezes e para diferentes locais de memória, é possível despejar toda a memória de um computador [32]. Desta forma, o atacante pode usar a cache como uma espécie de canal secreto para transferir informações sobre a memória acedida ilegalmente. Apesar do caminho especulativo ter sido apagado, é guardada uma "impressão digital" do conteúdo da memória ilegal na cache.

Os ataques Meltdown/Spectre podem ser classificados como uma fuga de informação e não existe, atualmente, uma forma de a evitar através de análise estática, pois todo o processo é baseado na forma como os processadores são construídos. Por outro lado, todos os restantes casos referidos anteriormente, poderiam ter sido evitados através de análise estática.

A autodestruição do Ariane 5 deveu-se à perda completa de informação de orientação e altitude logo após o seu lançamento. Esta perda de informação ocorreu devido à falha do Sistema de Referência Inercial (SRI), responsável por rastrear o movimento do foguete, devido a um erro de transbordamento (*overflow*). Especificamente, este erro ocorreu numa tentativa de conversão da velocidade lateral do foguete, que correspondia a um número de vírgula flutuante de 64 bits, para um número inteiro com sinal de 16 bits. Normalmente, as conversões de dados num programa estão protegidas de forma a identificar e recuperar dos possíveis erros que possam ocorrer. Na verdade, muitas das conversões de dados na programação do sistema de orientação do Ariane 5 incluíam essa proteção. Mas, neste

caso específico, os programadores decidiram que o valor da velocidade lateral do foguete nunca seria grande o suficiente para causar problemas [29]. Portanto, este acidente, que custou 10 anos de pesquisa e US $7 bilhões à Agência Espacial Europeia, poderia ter sido evitado se o erro de transbordamento, que esteve na sua origem, estivesse a ser tratado pelo software, impedindo que o SRI parasse de funcionar. Neste caso, havia conhecimento de que um erro podia existir, no entanto, existem casos em que esses cenários não são conhecidos. Utilizando análise estática é possível identificar cenários não considerados pelos programadores, evitando a ocorrência de erros como o que causou a autodestruição do Ariane 5.

O erro de cálculo identificado nos processadores Pentium 5 da Intel estava relacionado com um defeito na própria unidade de ponto flutuante (FPU). Devido a este defeito, o processador retornava valores errados para certas operações de divisão [49]. Mais um vez, este defeito tinha sido identificado ainda na fase de testes, mas foi ignorado, porque os testes realizados mostravam que se tratava de um erro muito raro. A Intel declarou que um utilizador comum só perceberia o erro uma vez a cada 27,000 anos. Mais tarde, pesquisas realizadas pela IBM revelaram que, na realidade, a probabilidade do erro ocorrer era de uma vez a cada 24 dias [66]. Portanto, os testes realizados pela Intel levaram a uma conclusão errada em relação à frequência de ocorrência do erro. A utilização de análise estática poderia ter contribuído para uma melhor estimativa da ocorrência deste erro, na medida em que seriam analisados caminhos lógicos não cobertos pelos testes.

O caso da destruição do Mars Climate Orbiter (MCO) na atmosfera de Marte ocorreu devido a um erro de navegação quando a nave, que o transportava, tentava efetuar a sua inserção na órbita deste planeta. Este erro ocorreu devido a uma inconsistência entre as unidades utilizadas pelos softwares envolvidos na manobra. O software terrestre, responsável por calcular os parâmetros para a manobra de inserção orbital, usava valores em unidades imperiais, enviando-os à nave, cujos sistemas apenas realizavam cálculos em unidades métricas. Assim, esses valores foram interpretados de forma errada e a nave entrou na órbita de Marte numa trajetória que a aproximou demasiado do planeta, fazendo com que passasse pela sua atmosfera superior e se desintegrasse. Mais uma vez, este acidente poderia ter sido evitado se tivesse sido utilizada uma forma de validação dos valores enviados pelo software terrestre para o software da nave, de forma a garantir a consistência das unidades utilizadas. De facto, após ocorrer este acidente, foi elaborada uma lista de recomendações para evitar que o mesmo pudesse acontecer à nave que transportava o Mars Polar Lander (MPL), que fazia parte da mesma missão do MCO e que ainda não tinha feito a sua manobra de inserção. Nessa lista, constavam recomendações que envolviam a verificação do uso consistente das unidades em todas as operações e *design* da nave e realizar revisões independentes sobre todos os eventos de missão crítica, entre muitas outras [67].

Por fim, o bloqueio do terminal 5 no Aeroporto de Londres-Heathrow foi atribuído a vários problemas no software que, em apenas 5 dias, causaram o extravio de mais

de 23.000 malas, o cancelamento de 500 voos e a perda £16 milhões por parte da British Airways (BA). Os problemas de software foram considerados, pela BA, como uma consequência do atraso nos trabalhos de construção, que impediu que fossem realizados todos os testes necessários antes da abertura do terminal [70]. A utilização de análise estática durante a fase de desenvolvimento do software garante que alguns erros sejam identificados mais cedo e que, portanto, a validação e verificação do software não fique dependente inteiramente dos testes.

Note-se que, para além do contributo na deteção de erros, a análise estática poderia também ser utilizada de forma a evitar os vários problemas referidos através do sistema de tipos ou anotando o código com asserções.

1.2 Identificação do problema

> *"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."*
>
> (Edsger Dijkstra,The Humble Programmer, ACM Turing Lecture 1972)

Atualmente, os testes desenvolvidos para assegurar a correção do software são muito exaustivos e dispendiosos. No entanto, como afirmado por Dijkstra, os testes não provam a ausência de erros, ou seja, a impossibilidade do software falhar em alguma circunstância não considerada. Idealmente, queremos conhecer o comportamento do nosso programa em todos os caminhos lógicos que este pode tomar durante a sua execução.

Erros como a fuga de memória, desreferências inválidas, operações inválidas de libertação de memória e acesso a variáveis não inicializadas podem causar comportamento indefinido nos programas. O comportamento indefinido existe em linguagens de programação inseguras, como é o caso da linguagem C. A segurança de uma linguagem de programação depende do tempo que esta demora a captar um erro. Quanto mais cedo um erro for identificado, mais segura é a linguagem. Numa linguagem de programação insegura, depois da execução de uma operação errada, o programa pode continuar, mas de uma maneira silenciosamente defeituosa que pode ter consequências observáveis mais tarde. Por este motivo, nestas linguagens, as operações erradas dizem ter comportamento indefinido [59]. A linguagem C é uma linguagem insegura, porque os seus criadores queriam que esta fosse uma linguagem de programação de baixo nível extremamente eficiente. A Lista de Questões Frequentes (FAQ) da linguagem de programação C define o comportamento indefinido da seguinte forma:

> "Qualquer coisa pode acontecer; O Padrão não impõe requisitos. O programa pode falhar ao compilar ou pode ser executado de forma incorreta (falhando ou gerando silenciosamente resultados incorretos), ou pode fortuitamente fazer exatamente o que o programador pretendia."

Ou seja, o programa pode continuar a funcionar normalmente ou pode falhar. Este tipo de falhas são de difícil deteção e reprodução, pelo que, em muitos casos, as circunstâncias em que os erros causados por comportamento indefinido ocorrem não são cobertas pelos testes do software.

É essencial fazer a verificação e validação do software durante toda a sua fase de desenvolvimento. Sendo que, a verificação tem como objetivo garantir que o software é desenvolvido de forma correta, isto é, se satisfaz os requisitos estabelecidos na sua especificação. Por outro lado, a validação garante que o software é o correto, ou seja, se satisfaz os requisitos impostos pelo cliente. As técnicas utilizadas para fazer a verificação e validação devem ser adequadas ao tipo de software que queremos analisar.

Atualmente, a verificação e validação de software é feita, essencialmente, através de três técnicas: revisão manual de código, análise semi-automática (provadores automáticos de teoremas [20]) e análise automática (dinâmica [6] ou estática [76]). A revisão manual de código é uma técnica realizada por seres humanos, portanto a sua confiabilidade está dependente dos mesmos. Além disso, esta técnica é demorada e tem custos muito elevados. De facto, em projetos de grandes dimensões, a revisão manual de código é infazível, porque é impossível rever a totalidade do projeto em tempo útil.

A análise semi-automática é realizada utilizando provadores automáticos de teoremas. Esta técnica, apesar de ser bastante poderosa, exige demasiado trabalho por parte dos programadores, pois é necessário anotar e alterar o código fonte. Além disso, exige que os programadores tenham um conhecimento profundo de lógica.

A análise dinâmica funciona de duas formas: integrando o código introspetivo numa aplicação em tempo de construção ou fornecendo uma forma de emulação de plataforma para entender o comportamento interno de uma aplicação em tempo de execução. À medida que a dimensão e o tempo de execução do programa aumenta, o desempenho das ferramentas de análise dinâmica vai-se degradando, pois a construção dos modelos necessários é uma tarefa que escala com a dimensão do software e tempo de execução, chegando a atingir dimensões que não cabem em memória [65].

Finalmente, a análise estática é uma técnica que examina o código fonte, ou uma representação intermédia do mesmo, sem que este seja executado. O facto de não ser necessário executar o programa permite que esta técnica possa ser utilizada durante toda a fase de desenvolvimento do software. A utilização precoce de análise estática permite a identificação de erros numa fase inicial do desenvolvimento, o que reduz consideravelmente os custos. Além disso, o processo realizado pelas ferramentas de análise estática calcula todos os valores de variáveis para todos os possíveis caminhos lógicos que o programa pode percorrer e não apenas os percorridos durante uma execução em particular. Esta característica da análise estática é especialmente importante para assegurar a segurança do software, uma vez que os ataques de segurança focam-se, frequentemente, em executar o programa de uma forma imprevista e não testada [21].

As ferramentas de análise estática utilizam, frequentemente, sobre-aproximação das

propriedades que pretendem verificar. Este processo tem como consequência a identificação de falsos positivos (i.e., erros que não existem). Os falsos positivos requerem um tratamento adicional que tem custos não negligenciáveis e cuja filtragem automática pode facilmente gerar falsos negativos, ou seja, erros reais que não são reportados. A grande quantidade de erros reportados pelas ferramentas de análise estática é o maior obstáculo à sua utilização no contexto industrial [46].

Interessa-nos que a verificação do software seja feita da forma mais automática possível, isto é, com pouca necessidade de intervenção por parte do programador. De facto, a verificação automática do software tem sido um objetivo de longa data na área da ciência da computação. Para além de exigir menos esforço por parte do programador, a automatização da verificação do software permite que este processo seja mais rigoroso e completo, uma vez que é baseado em teorias académicas, tais como a lógica de Hoare [34] e a interpretação abstrata [14]. Isto é, as propriedades estão definidas matematicamente e, portanto, são identificadas todas as situações em que as mesmas são violadas. No entanto, a integração de uma teoria académica numa ferramenta automática exige muito esforço. O Infer [10], também conhecido como "Facebook Infer", é um caso de sucesso desta integração. Esta ferramenta é, atualmente, desenvolvida e utilizada no contexto do Facebook e baseia-se em pesquisas académicas na área da ciência da computação, nomeadamente na lógica da separação [60], uma extensão da lógica de Hoare, e na interpretação abstrata [50]. Por uma opção de engenharia, o Infer não utiliza todo o poder da lógica de separação (que além de indecidível tem complexidade computacional elevada e exige profundo conhecimento teórico), uma vez que se pretende que esta seja automática e decidível, para ser útil aos programadores. Desta forma, o Infer é uma ferramenta de análise estática útil, eficiente e que não exige que os utilizadores tenham um conhecimento profundo sobre lógica de separação.

Concluindo, consideramos que a utilização de ferramentas de análise estática é a opção que mais se adequa à verificação de software. Esta técnica permite verificar o código sem exigir muito esforço do programador e de forma automática, rigorosa e completa.

1.3 Objetivo

O objetivo desta dissertação é fazer uma análise comparativa de várias ferramentas automáticas de análise estática aplicadas a um conjunto de softwares (programas) escritos na linguagem C/C++, com diferentes dimensões e complexidades.

O objetivo não é decidir qual das ferramentas é melhor, uma vez que cada ferramenta oferece funcionalidades diferentes. Até mesmo quando as ferramentas aparentemente fornecem a mesma funcionalidade, o tipo de teoria na qual são baseadas não é comparável. Em vez disso, será feita uma análise das funcionalidades de cada uma das ferramentas, com foco na sua capacidade de identificação de erros de memória, um dos erros mais comuns da linguagem C/C++.

O objetivo principal divide-se nos seguintes sub-objetivos:

1. Identificar e selecionar ferramentas automáticas de análise estática;
2. Identificar e selecionar o conjunto de projetos a ser analisados;
3. Analisar o histórico de erros dos projetos selecionados;
4. Selecionar as versões relevantes a serem analisadas em cada um dos projetos;
5. Aplicar as ferramentas selecionadas nas diferentes versões dos projetos;
6. Validar os relatórios de resultados das ferramentas;
7. Identificar padrões de erros;
8. Construir exemplos mínimos dos padrões identificados;
9. Elaborar um relatório de comparação das ferramentas.

Os sub-objetivos enumerados são detalhados nas secções do Capítulo 3.

1.4 Principais contribuições

As principais contribuições desta dissertação são os seguintes:

- Compreensão do grande potencial e o elevado impacto que as ferramentas de análise estática podem ter na validação e verificação de software.
- Desenvolvimento de uma metodologia de trabalho que pode ser seguida em estudos similares.
- Identificação das funcionalidades das ferramentas analisadas, através do reconhecimento de padrões de erros.
- Identificação e reporte de novos erros em programas de código aberto e com elevada popularidade.
- Construção de exemplos mínimos capazes de reproduzir os resultados observados nos programas analisados.

1.5 Organização do documento

Este documento está organizado segundo a seguinte estrutura:

- Capítulo 1 – Introdução: Contextualização, identificação do problema, objetivo e principais contributos da dissertação.
- Capítulo 2 – Trabalho relacionado: Apresentação de conceitos e breve descrição de trabalhos anteriores relacionados com o tema desta dissertação.

- Capítulo 3 – Metodologia de trabalho: Descrição das várias etapas seguidas durante a elaboração desta dissertação.
- Capítulo 4 – Experimentação e Análise: Apresentação e análise dos resultados obtidos da execução das ferramentas de análise estática sobre os vários projetos.
- Capítulo 5 – Uma comparação das ferramentas: Descrição detalhada dos pontos diferenciadores das várias ferramentas estudadas.
- Capítulo 6 – Conclusões: Síntese da trabalho desenvolvido e apresentação das lições retiradas do estudo realizado nesta dissertação.

Capítulo 2

Trabalho relacionado

No Capítulo 1 foi feita uma introdução ao contexto e problema desta dissertação, na qual foi referida a importância de garantir a ausência de erros no código. Como foi discutido nesse capítulo, a opção por validação com base em testes apresenta algumas limitações e não consegue cobrir todas as circunstâncias em que um erro pode ocorrer.

No presente capítulo são apresentadas técnicas de validação de software que, ao contrário dos testes, são capazes não só de identificar erros no código, mas também provar a sua ausência. A primeira dessas técnicas é a revisão manual de código. Esta técnica pode ser utilizada, seguindo várias metodologias, para encontrar possíveis problemas no código durante a fase de desenvolvimento do software, enquanto ainda é barato corrigir esses erros. No entanto, a revisão de código manual não consegue identificar todos os tipos de erros e as circunstâncias em que estes podem ocorrer, pois é realizada por seres humanos. Além disso, a revisão manual de código é uma técnica demorada e com muitos custos. A segunda técnica é a análise estática, que consiste numa revisão automática de código realizada por ferramentas. Este tipo de análise pode, com o tipo certo de aproximações, verificar todas as possíveis execuções do programa e fornecer garantias sobre as suas propriedades.

Ainda neste capítulo são apresentados vários trabalhos relacionados com o tema desta dissertação: i) QualityAssistant [72]; ii) Mute [68]; iii) AddressSanitizer [63]; iv) Renraku [73]; v) Repositório de alertas de análise estática [46]; Muitos destes trabalhos têm como objetivo encontrar uma forma de filtrar a grande quantidade de erros devolvida pelas ferramentas de análise estática. Outros, por outro lado, são agregações de várias ferramentas, que tentam, desta forma, obter resultados mais completos e com menos falsos positivos.

2.1 Revisão de código

A revisão de código consiste numa técnica na qual o código desenvolvido vai ser sujeito a uma análise, manual ou automática, com o objetivo de garantir e/ou melhorar a qualidade do software. A revisão de código não tem apenas como objetivo identificar erros no código, mas também promover boas práticas de programação, expor vulnerabilidades e verificar se existe *malware*. Os revisores revêm o código escrito por outros programadores e dão-lhes *feedback* sobre o mesmo, mas nunca podem rever o seu próprio código nem alterar o código que estão a analisar. O objetivo de um revisor é encontrar problemas no código e não encontrar solução para esses problemas.

Os princípios SOLID constituem um exemplo de regras que podem ser seguidas durante uma revisão de código. Estes princípios consistem em 5 regras de *design* que têm com objetivo promover a flexibilidade, manutenção e compreensão do software. A descrição de cada um destes princípios encontra-se na Tabela 2.1.

Tabela 2.1: Princípios SOLID.

Princípio	**Descrição**
Responsabilidade Única	Uma classe deve ter apenas uma única responsabilidade, sendo que todos os seus métodos devem estar relacionados com essa responsabilidade [9].
Aberto-Fechado	As entidades de software devem estar abertas para extensão, mas fechadas para modificação [43].
Substituição de Liskov	Os objetos num programa devem poder ser substituídos por instâncias dos seus subtipos sem alterar a correção do programa [42].
Segregação de Interfaces	Os clientes não devem ser forçados a depender de métodos que não usam, portanto, as interfaces devem ser pequenas e específicas para que os clientes só tenham que saber sobre os métodos que lhes interessam. Ter muitas interfaces específicas para cada cliente é melhor que ter uma única interface geral [41].
Inversão de Dependência	Os módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. Sendo que, as abstrações não devem depender dos detalhes, mas sim o inverso, ou seja, os detalhes devem depender das abstrações [40].

Muitas das verificações de código podem ser feitas através de ferramentas automáticas, no entanto, existem ainda alguns aspetos que necessitam de ser revistos manualmente. Existem coisas em que os seres humanos são bons a identificar numa revisão de código

e outras em que uma ferramenta será mais útil. Portanto, durante uma revisão manual de código, devemos focar-nos nas coisas que somos bons a identificar, como por exemplo problemas relacionados com o *design*, legibilidade, manutenção e funcionalidade [27].

As vantagens de incluir a revisão de código no processo de desenvolvimento do software incluem [24]: i) Bom retorno sobre investimento; ii) Encontrar problemas no código mais cedo; iii) Poupar tempo e reduzir os custos; iv) Equipas mais eficientes; v) Melhoria dos testes; vi) Podemos aplicar os mesmos métodos a projetos futuros.

Uma revisão de código começa pelo planeamento da mesma, no qual vão ser decididas quais as partes do código a rever e que técnicas devem ser aplicadas. As técnicas mais comuns de revisão manual de código são as seguintes [24]: i) Revisão regular; ii) Revisão baseada em obrigações; iii) Revisão baseada em padrões de erros ou lista de verificações; iv) Revisão baseada em *verificação de secretária*[1]; v) Técnica de revisão intercalada (IRT);

Estas técnicas podem ser usadas isoladamente ou conjugadas para se obterem melhores resultados. Nas secções seguintes é feita uma pequena descrição de cada uma das técnicas listadas.

2.1.1 Revisão Regular

A revisão regular, também conhecida como revisão de Fagan, foi formalizada pela primeira vez na década de 70 pela IBM. Esta técnica tem como objetivo encontrar defeitos no código durante as várias fases do processo de desenvolvimento do software. Os defeitos são identificados com base nos critérios de entrada e saída especificados para uma determinada operação ou processo. Isto é, o revisor vai validar se o resultado do processo está em conformidade com os seus critérios de saída. As revisões de código que seguem esta técnica começam pelo planeamento da revisão, que deve ser feito pelo moderador do processo. Seguidamente, o autor vai entregar o código ao revisor, que vai fazer a sua análise. Depois da revisão estar concluída, realiza-se uma reunião para discutir os problemas encontrados no código e que alterações devem ser feitas. Após as alterações estarem concluídas voltamos ao início do processo e é feita uma nova revisão. Quando o moderador considerar que não são necessárias mais alterações, o processo de revisão de código é dado como concluído. Com base na revisão Fagan, muitas metodologias de revisão têm sido sugeridas. De facto, os elementos que constituem a revisão Fagan são utilizados em todas as técnicas de revisão, que são descritas de seguida [24].

2.1.2 Revisão baseada em Obrigações

Esta técnica é baseada numa lista de obrigações definidas para o sistema que está a ser revisto. Uma obrigação diz o que deve e não deve acontecer antes e depois de ser feita uma chamada a esse sistema. Durante a revisão do código, a obrigação de cada chamada deve ser definida e verificada para todos os caminhos lógicos. Para facilitar a revisão podem

[1] Tradução para Desk Checking Review.

ser usadas asserções. Neste tipo de revisão vamos ter alguém responsável por estabelecer quais as obrigações de cada chamada ao sistema e vários revisores que verificam se as obrigações estão a ser cumpridas em todos os caminhos lógicos possíveis [24]. Neste tipo de revisão, tal como em qualquer revisão manual, a confiabilidade do processo está dependente de seres humanos, para que haja uma verificação rigorosa do código é necessário usar provadores automáticos.

2.1.3 Revisão baseada em Padrões de Erros ou Lista de Verificações

Nesta técnica são definidos quais os padrões de erros que se querem evitar ou quais são as regras de estilo e funcionalidade que o código deve seguir. As regras definidas são organizadas numa lista de verificações, que será seguida durante o processo de revisão de código. Cada lista de verificações deve estar direcionada para a área correta, pois muitas vezes as listas pré-definidas focam-se mais em questões de estilo e menos nos possíveis erros. Neste tipo de revisão os revisores vão verificar se o código cumpre com todos os pontos presentes na lista. Sendo que, numa revisão baseada em lista de verificações, muitas vezes são identificadas partes do código que não representam necessariamente um erro, mas sim uma violação de um dos parâmetros presentes na lista [24].

2.1.4 Revisão baseada em Verificação de Secretária

A revisão baseada em *verificação de secretária* consiste numa simulação da execução do programa que está a ser revisto. É uma técnica bastante eficaz na deteção de erros na fase inicial do desenvolvimento e é facilmente aplicada a código sequencial. Nesta técnica é feita uma verificação da lógica dos algoritmos que constituem o programa, de forma a identificar erros que podem impedir que este funcione como esperado. A primeira fase desta revisão é a criação de cenários de teste. Após esta fase, é feita uma execução manual do código para cada um dos cenários construídos. Normalmente, os resultados do processo de execução manual são representados numa tabela com várias colunas contendo os números das linhas do código, variáveis, condições, entradas e saídas. Observando a tabela criada é possível saber os valores que as variáveis tomam em cada passo da execução e, portanto, conseguimos identificar erros se algum desses valores não for o esperado [24].

2.1.5 Técnica de Revisão Intercalada

Esta técnica é uma derivação da revisão baseada em *verificação de secretária* e é orientada para concorrência e tolerância ao erro. O objetivo da técnica de revisão intercalada é verificar todos os possíveis cenários que podem ocorrer no sistema concorrente ou distribuído em revisão. O estado de um sistema concorrente é determinado pelos estados de todos os seus processos e dependências, portanto, torna-se muito mais difícil seguir o fluxo de dados no código e recriar possíveis falhas. Por exemplo, se existirem 3 processos com 10 estados cada um, existem 1000 cenários possíveis para revisão. Por estes motivos, a

criação de testes para sistemas concorrentes é muito mais cara do que para um sistema de código sequencial. Na técnica de revisão intercalada a construção dos cenários de teste é feita através do produto cartesiano dos escalonamentos possíveis de partes aleatórias do código do programa. Ou seja, ao contrário da revisão baseada em *verificação de secretária*, a execução manual do código não é feita de forma sequencial, em vez disso, são executadas partes aleatórias do código segundo um determinado escalonamento (i.e., cada cenário de teste corresponde a um escalonamento). Desta forma, são considerados novos cenários de teste e a probabilidade de encontrar um erro é maximizada [26].

2.2 Análise estática

A revisão manual de código tem algumas desvantagens, nomeadamente, o facto de ser realizada por seres humanos, o que implica que seja um processo falível, dispendioso e demorado. Na realidade, quando o projeto tem uma dimensão muito grande, a revisão manual é infazível. Além disso, a confiabilidade desta técnica está totalmente dependente dos seres humanos, sendo natural existirem falhas na revisão e cenários de teste que não são considerados. Devido à baixa eficiência e confiabilidade, a utilização de revisão manual de código pode não ser suficiente, é necessário utilizar técnicas complementares de forma a tornar o processo de revisão mais eficiente, rigoroso e completo. Uma dessas técnicas é a análise estática.

A análise estática pode ser definida como o conjunto de algoritmos e técnicas aplicados sobre uma representação do código do programa, que queremos analisar, sem que este seja executado. A análise estática é um processo automático e, geralmente, é realizada como parte da revisão de código durante a fase de desenvolvimento do software. A sua utilização permite a identificação de erros numa fase inicial do desenvolvimento, o que reduz consideravelmente os custos, uma vez que a descoberta de defeitos é exponencialmente mais dispendiosa ao longo do tempo.

As ferramentas de análise estática usam uma variedade de métodos que lhes permitem identificar vulnerabilidades no código. O método mais usual, que se "banalisou", é a utilização de sistemas de tipos. Interessa-nos estudar a viabilidade de usar ferramentas, nomeadamente para procurar erros como fuga de memória, desreferências e operações de libertação de memória inválidas, acesso a variáveis não inicializadas e até problemas de concorrência [65]. No entanto, a análise estática não é perfeita, uma vez que algumas das propriedades que esta tenta verificar são indecidíveis. Se ignorarmos as limitações da memória finita, podemos considerar que as linguagens de programação utilizadas no desenvolvimento do software são Turing completas. Uma linguagem de programação Turing completa é teoricamente capaz de expressar todas as tarefas realizáveis pelos computadores, isto é, tem o mesmo poder de processamento de uma máquina de Turing [71]. Esse poder de processamento corresponde ao de todos os computadores atuais, não existindo nenhum computador comercial com um poder de processamento maior que o de uma máquina de Turing. Um problema é resolvido para uma máquina de Turing se e só

se for resolvido para um computador. Em 1936, Alan Turing, através da sua definição matemática das máquinas de Turing, provou que um algoritmo geral capaz de resolver o problema da paragem[2], para todos os possíveis programas com todos os possíveis inputs, não pode existir, ou seja, é um problema indecidível. Desta forma, foi demonstrado que nem todos os problemas podem ser resolvidos pelas máquinas de Turing, o problema da paragem é uma limitação destas máquinas. A indecidibilidade do problema da paragem foi generalizada mais tarde pelo teorema de Rice [61], que afirma o seguinte:

> "Qualquer propriedade não-trivial de uma linguagem reconhecida por uma máquina de Turing é indecidível."

Uma propriedade é considerada não-trivial se existir uma máquina de Turing que tenha essa propriedade, e pelo menos uma que não tenha. Na prática, isso significa que não existe uma máquina capaz de decidir sempre se a linguagem de uma dada máquina de Turing tem uma propriedade não-trivial [62]. O problema da paragem é uma propriedade não-trivial, na medida em que existem máquinas de Turing que param e outras, com ciclos infinitos, que nunca irão parar. Portanto, segundo o teorema de Rice, não é possível construir uma máquina capaz de decidir sempre se uma dada máquina de Turing vai parar ou não.

Os conceitos levantados pelos teoremas de incompletude de Gödel [30] são muito semelhantes aos levantados pelo problema da paragem, assim como as suas provas. Os teoremas da incompletude de Gödel foram os primeiros de vários teoremas estreitamente relacionados sobre as limitações dos sistemas formais. O primeiro teorema de Gödel pode ser indicado, de modo genérico, da seguinte forma [57]:

> Primeiro Teorema: "Qualquer sistema formal consistente F dentro do qual uma certa quantidade de aritmética elementar pode ser realizada é incompleta; ou seja, há declarações da linguagem de F que não podem ser comprovadas nem refutadas em F."

Sucintamente, o primeiro teorema de incompletude de Gödel afirma que para qualquer sistema formal suficientemente poderoso, existem afirmações que não podem ser provadas através desse sistema como sendo verdadeiras ou falsas. Uma analogia semântica deste teorema é o Paradoxo do Mentiroso, no qual é considerada a seguinte frase: "Esta frase é falsa". Uma análise desta frase mostra que não pode ser verdade, pois, como indica, é uma frase falsa. Por outro lado, também não pode ser considerada falsa, porque, nesse caso, a frase é verdadeira. Concluindo, não é possível classificar a frase como verdadeira ou falsa [57].

A partir da redução do problema da paragem e dos teoremas da incompletude de Gödel é possível concluir que alguns dos problemas da análise estática são indecidíveis,

[2] Dado um programa arbitrário e um input, é possível determinar se esse programa vai terminar?

pois as propriedades interessantes de um programa são, normalmente, indecidíveis. Portanto, a utilização de sobre-aproximação é inevitável [45]. Através da sobre-aproximação a análise estática consegue fornecer soluções úteis para problemas decidíveis que contêm os problemas indecidíveis. Isto é, a análise estática determina que decisão tomar em relação aos problemas indecidíveis com base na decisão tomada para os problemas decidíveis. Como consequência, essas ferramentas geralmente devolvem falsos positivos, ou seja, um resultado no qual existe uma violação de uma propriedade decidível, mas não da propriedade indecidível em que se está interessado.

O processo de análise estática, representado na Figura 2.1, começa por extrair o modelo, ou seja, transformar o código numa representação intermédia. Essa representação é uma estrutura de dados que é depois percorrida pela ferramenta durante a fase de análise do código. No final da fase de análise, a ferramenta devolve um relatório de resultados. No entanto, nem todos os processos de análise estática passam pela transformação do código fonte, realizando a sua análise diretamente sobre o mesmo. Um exemplo deste tipo de abordagem é o sistema de tipos, que consiste num conjunto de regras que atribuem um tipo às várias construções que constituem um programa (i.e. variáveis, expressões, funções ou módulos). Assim, são criadas interfaces entre as várias partes do programa que vão depois ser usadas pelas ferramentas de análise estática para verificar a existência de erros no código. A consistência entre as várias interfaces do programa é o que garante a ausência de erros de tipo no código analisado, como por exemplo, operações de divisão sobre strings ou variáveis do tipo booleano. O sistema de tipos, tal como as restantes abordagens de análise estática, utiliza sobre-aproximação, logo gera falsos positivos.

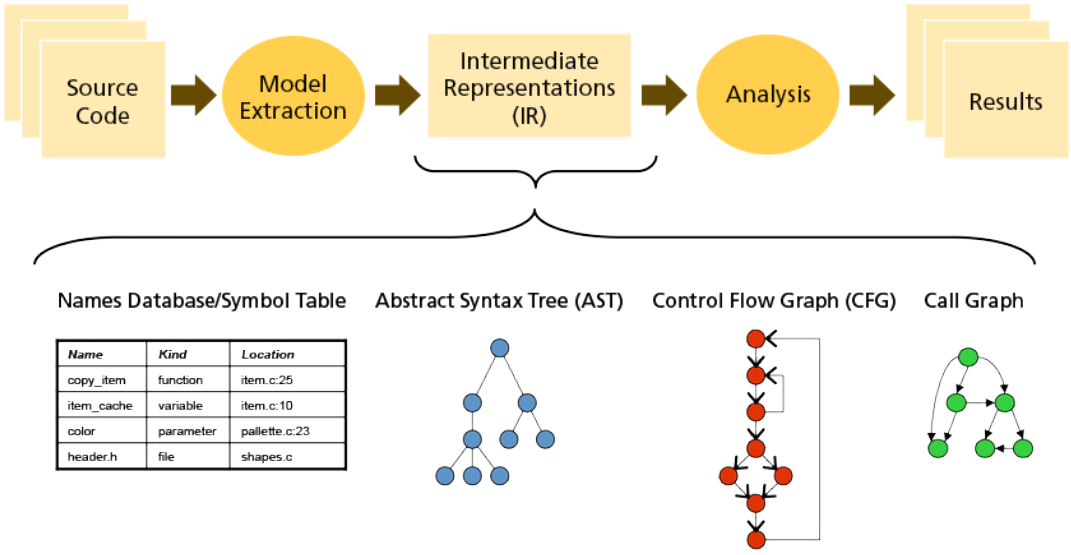


Figura 2.1: Processo de análise estática.
(Figura retirada de [37])

Atualmente, as ferramentas de análise estática utilizam diferentes tipos de abordagem nos seus processos de análise do código. Algumas das abordagens para a identificação dos erros em que estamos interessados são as seguintes:

Interpretação Abstrata (AI): abordagem que serve para modelar o efeito que cada declaração tem sobre o estado de uma máquina abstrata. Isto é, dada uma linguagem de programação, a interpretação abstrata consiste em atribuir várias semânticas ligadas por relações de abstração. Uma semântica é uma caracterização matemática do possível comportamento do programa. Portanto, o programa é executado com base nas propriedades matemáticas de cada declaração. Nem todas as propriedades verdadeiras do sistema original são verdadeiras no sistema abstrato, portanto, o sistema abstrato é mais simples de analisar devido à sua incompletude. Se esta abordagem for aplicada corretamente, todas as propriedades verdadeiras do sistema abstrato podem ser mapeadas para uma propriedade verdadeira do sistema original [14].

Árvore de Sintaxe Abstrata (AST): uma representação abstrata em árvore do código-fonte de um programa, na qual as folhas representam as constantes e as variáveis e os nós internos representam os operadores ou declarações [47].

Grafo de Controlo de Fluxo (CFG): , que consiste num grafo orientado, no qual são representados todos os caminhos lógicos que podem ser percorridos durante a execução de um programa [3].

Lógica de Separação (SL): uma lógica matemática, que consiste numa extensão da lógica de Hoare [34] para o raciocínio sobre programas que acedem e alteram dados mantidos em estruturas de dados dinâmicas. Esta lógica permite a escalabilidade ao partir o raciocínio em várias partes, correspondentes às operações locais na memória e, seguidamente, juntar as partes do raciocínio novamente [60].

Grafo de Memória Abstrata (SMG): grafo direcionado que representa simbolicamente a memória e que é composto por objetos (espaço alocado), valores (endereços) e arestas que ligam os vários elementos do grafo. Assim, cada bloco de memória utilizado pelo programa dá origem a um SMG [19].

Nas secções seguintes são descritas abordagens alternativas à seguida nesta dissertação. Sendo que, estas têm como objetivo melhorar o desempenho das ferramentas de análise estática.

2.2.1 QualityAssistant

No Artigo [72] de Tymchuk et al. (2016) é apresentado o QualityAssistant. Este "assistente" consiste numa série de *plug-ins* que usam as regras presentes na ferramenta SmallLint[3], para ferramentas de desenvolvimento com o objetivo de fornecer um *feedback* intrusivo

[3] http://www.refactory.com/smalllint

sobre a qualidade do código-fonte. As ferramentas de análise estática são úteis para a verificação de software, no entanto, são ainda pouco utilizadas no contexto industrial. Os programadores sentem a necessidade de personalizar a maneira como as ferramentas funcionam ou de alterar as regras que são usadas na análise do código-fonte. O QualityAssistant permite aos programadores realizar as alterações desejadas.

A utilização do QualityAssistant desencadeou um ciclo de *feedback* que levou à melhoria ou remoção das regras existentes e à integração de novas regras nas ferramentas [72]. A análise realizada destas mudanças sugere que regras precisas que capturam uma lógica específica de domínio são mais relevantes que as regras gerais. Assim, os *plug-ins* instruíram os programadores sobre a existência de regras e problemas de qualidade nos seus códigos. A análise das regras adicionadas e removidas ajuda-nos a identificar os recursos que são importantes para os programadores: i) Explicação clara que expõe a fonte do erro; ii) Sugestão de solução ou resolução automatizada de um problema; iii) Alto impacto da ferramenta, ou seja, as ferramentas que detetam erros reais são mais valorizadas.

O QualityAssistant teve uma aceitação muito positiva, sendo que em combinação com a facilidade de atualização do SmallLint conduziu a muitas alterações nas regras de qualidade. Apesar da grande utilidade deste recurso não se considerou utiliza-lo durante a elaboração desta dissertação, pois o foco da mesma são as ferramentas de análise estática sem a aplicação de *plug-ins* adicionais.

2.2.2 Mute

No Artigo [68] de Teixeira et al. (2007) é apresentada uma nova ferramenta, designada por Mute. Esta ferramenta utiliza um mecanismo de agregação de resultados produzidos por várias ferramentas de análise estática.

O processo descrito no artigo começou pela definição de um conjunto extensivo de classes de vulnerabilidades, que inclui problemas como a validação de parâmetros e a conversão explícita de tipos. Seguidamente, foi desenvolvido um programa com vários exemplos destas classes de vulnerabilidades e alguns excertos de código corretos. Este programa foi entregue a 9 ferramentas para análise: i) Crystal; ii) Deputy; iii) Flawfinder; iv) ITS4; v) MOPS; vi) PScan; vii) RATS; viii) Sparse; ix) UNO [35]; Os excertos foram utilizados para determinar a quantidade de falsos positivos gerada pelas ferramentas. Após obtidos os relatórios procedeu-se à comparação das ferramentas, feita com base no cálculo da eficácia e da precisão. Assim, observou-se que as ferramentas encontram-se normalmente especializadas para a deteção de certas classes específicas de vulnerabilidades, não havendo nenhuma capaz de localizar um grande número de classes. Além disso, verificou-se que as ferramentas devolvem uma quantidade considerável de falsos positivos. Depois de concluída a análise de desempenho das ferramentas, procedeu-se à construção do Mute. Resumidamente, o Mute recolhe os resultados de análise de cada ferramenta e formata-os num modelo pré-definido. A partir daqui, agrega os resultados de acordo com um algoritmo e decide que vulnerabilidades devem ser reportadas ao

utilizador. Foram considerados dois algoritmos de decisão diferentes, sendo que ambos se baseiam em observar qual das ferramentas identificou o erro e quantas foram capazes de o identificar. O primeiro algoritmo consiste em atribuir um nível de confiança a cada ferramenta. O segundo algoritmo consiste em atribuir um nível de confiança a cada classe de vulnerabilidades. Ou seja, cada ferramenta tem associada um vector de valores de confiança. A avaliação do Mute demonstrou a utilidade de se usarem métodos de agregação, tendo ele conseguido localizar mais vulnerabilidades do que as ferramentas de forma individual e, simultaneamente, devolver menos falsos positivos.

A análise dos resultados das ferramentas realizada no âmbito do desenvolvimento do Mute é semelhante ao elaborado nesta dissertação, diferindo apenas no projeto utilizado. Por um lado, no Mute foi desenvolvido um projeto com as características desejadas. Por outro lado, nesta dissertação as ferramentas foram aplicadas a projetos já existentes disponíveis no GitHub. Além disso, a quantidade de vulnerabilidades verificadas é mais reduzida, nesta dissertação o foco são os erros de memória. Finalmente, não se considerou a utilização do Mute, pois esta plataforma agrega várias ferramentas e o objetivo do presente estudo é comparar as várias ferramentas em separado.

2.2.3 Renraku

A maioria dos analisadores estáticos são aplicativos monolíticos que definem as suas próprias formas de analisar o código-fonte e apresentar os resultados. Portanto, a agregação de vários analisadores estáticos numa única ferramenta ou a integração de um novo analisador em ferramentas existentes exige uma quantidade significativa de esforço. No Artigo [73] de Tymchuk et al. (2018) é apresentado o Renraku, um modelo de análise estática que atua como um mediador entre os analisadores estáticos e as ferramentas que apresentam os relatórios. O Renraku pretende confirmar a hipótese de que um único modelo que forneça informações sobre as várias propriedades do código-fonte pode facilitar o desenvolvimento de mecanismos de análise e a sua integração em ferramentas de desenvolvimento. Assim, os principais objetivos deste modelo são reduzir o custo de: i) Fornecer relatórios de análise de código fonte personalizados em ferramentas de desenvolvimento existentes; ii) Obter e reutilizar as propriedades do código fonte fornecidas pelos analisadores disponíveis. Atualmente, o Renraku é usado principalmente pelo sistema de *feedback* de análise estática chamado QualityAssistant. Existem protótipos construídos para demonstrar a flexibilidade desta plataforma em combinação com várias ferramentas.

Concluindo, quando usado por programadores de análise e ferramentas, o Renraku pode reduzir o custo para introduzir um novo tipo de análise nas ferramentas existentes e criar uma ferramenta que se baseia em analisadores existentes. Novamente, o Renraku é um modelo e não uma ferramenta de análise estática. Portanto, apesar das reconhecidas vantagens a utilização deste modelo, a sua utilização não foi considerada para o estudo elaborado nesta dissertação.

2.2.4 Repositório de alarmes de análise estática

A grande quantidade de erros reportados pelas ferramentas de análise estática é o maior obstáculo à utilização destas ferramentas no contexto industrial. Além disso, as ferramentas de análise estática normalmente reportam os erros nos locais onde é provável que estes ocorram em tempo de execução e não na sua origem. Portanto, o programador tem que percorrer o código até encontrar a causa do erro e verificar se é real ou não. Dada elevada dimensão e complexidade do código industrial, este processo é muito difícil e demorado. No Artigo [46] de Muske et al. é apresentado um processo designado Reposicionamento de Alarmes, que tem como objetivo superar as limitações descritas. Esta técnica é automática e consiste num pós-processamento que move os alarmes para cima ou para baixo no fluxo de controlo do programa, sem afetar os erros descobertos. Sintetizando, os objetivos do reposicionamento são: i) Reduzir o número de alarmes reportados sem afetar os erros descobertos; ii) Reportar os alarmes mais próximos das suas causas.

A redução no número de alarmes é obtida movendo grupos de alarmes relacionados ao longo do fluxo de controlo. Os alarmes são movidos até um ponto do programa onde estes podem ser substituídos por um único alarme e são mantidos links entre estes dois locais. A técnica apresentada é independente de ferramentas e ortogonal a muitas outras técnicas disponíveis para o pós-processamento de alarmes. Esta técnica foi aplicada a 16 casos de estudo de código aberto e 4 comerciais tendo sido verificada uma redução de alarmes de 7,25%. Este resultado está no topo da redução de alarmes obtida através das técnicas de agrupamento. O Reposicionamento de Alarmes é uma técnica que pode ser aplicada sobre os relatórios de resultados das ferramentas de análise estática estudas. No entanto, o objetivo desta dissertação é realizar uma análise comparativa das ferramentas sem a aplicação de qualquer modelo de filtragem ou reposicionamento adicionais.

2.3 AddressSanitizer

No Artigo [63] de Serebryany et al. (2012) é apresentado o AddressSanitizer, uma nova ferramenta de análise estática, desenvolvida pela Google, para identificação de erros de memória. Os erros identificados por esta ferramenta incluem: acessos fora do limite da *heap*, pilha e objetos globais e utilização de memória após a sua libertação. A abordagem seguida pelo AddressSanitizer é similar à da ferramenta AddrCheck[4]. Isto é, usa a memória de sombra [48] para registar se é seguro aceder a cada um dos bytes de memória utilizados no programa e usa instrumentação para verificar o estado da memória de sombra após cada operação sobre a mesma. De notar que, a memória de sombra consiste numa técnica na qual são armazenadas informações na memória atribuída ao programa em execução. No entanto, o AddressSanitizer usa um mapeamento de sombra mais eficiente, uma codificação de sombra mais compacta. Assim, o AddressSanitizer alcança eficiência sem sacrificar a abrangência. A sua desaceleração média é de apenas 73%, mas identifica

[4]https://courses.cs.washington.edu/courses/cse326/05wi/valgrind-doc/ac_main.html

com precisão os erros no momento da ocorrência. Esta ferramenta foi responsável por identificar mais de 300 bugs desconhecidos no Google Chrome.

Esta ferramenta não foi utilizada no estudo realizado nesta dissertação, pois só se tomou conhecimento da mesma depois da seleção das ferramentas ter sido concluída. No entanto, tendo em consideração os resultados apresentados, considera-se que é uma ferramenta com potencial, que poderá ser avaliada em futuros estudos.

2.4 Síntese

Nesse capítulo é clarificado o facto de a revisão manual de código poder não ser suficiente para fazer a validação de software, uma vez que é um processo muito demorado, dispendioso e a sua confiabilidade depende dos seres humanos que o realizam. Portanto, é necessário utilizar técnicas complementares visando tornar o processo de revisão mais eficiente, rigoroso e completo. Devem ser incluídas técnicas automatizadas no processo de revisão de código, como é o caso da análise estática. No entanto, algumas das propriedades que esta técnica tenta verificar são indecidíveis. Por este motivo, as ferramentas de análise estática fazem sobre-aproximação e, como consequência, devolvem falsos positivos, o que obriga a fazer uma validação dos seus resultados. Essa validação é feita através da revisão manual de código. Concluindo, a revisão manual de código e a utilização de ferramentas de análise estática são técnicas complementares, que devem ser utilizadas em conjunto. Existem ainda várias abordagens que podem ser utilizadas para melhorar os resultados das ferramentas, tais como: i) QualityAssistant; ii) Mute; iii) Renraku; iv) Repositório de alarmes de análise estática.

Capítulo 3

Metodologia de trabalho

No Capítulo 2 foi feito um levantamento dos conceitos e técnicas da revisão de código e da análise estática. Neste capítulo são descritas as etapas seguidas durante o desenvolvimento desta dissertação, que se encontram representadas na Figura 3.1.

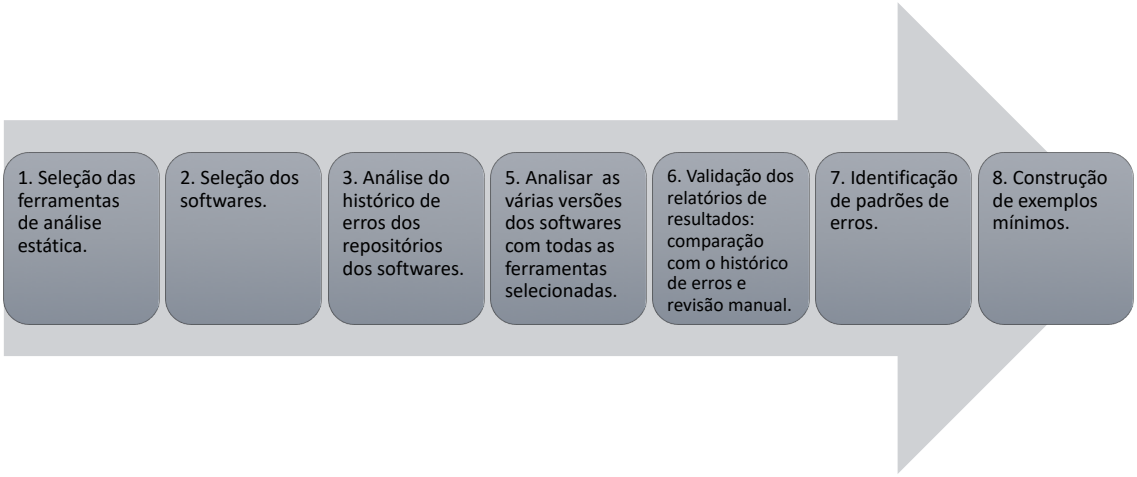


Figura 3.1: Etapas seguidas durante a realização do estudo comparativo.

O processo representado na Figura 3.1 inicia-se com a identificação e seleção das ferramentas de análise estática a estudar (Secção 3.1) e dos programas sobre os quais estas vão ser aplicadas(Secção 3.2). Concluída a fase de seleção das ferramentas e dos programas, seguiu-se a análise dos mesmos. Inicialmente, foram recolhidas informações dos repositórios dos programas sobre o número e tipo de erros de memória já identificados, ou seja, que fazem parte do histórico de erros daquele programa. Como forma de obter uma informação mais completa sobre os erros, foram analisados tanto os *commits*, como as questões colocadas pelos utilizadores no repositório (i.e., *issues*). Os erros reportados como *issue* foram verificados manualmente, de maneira a verificar se são erros reais ou

falsos positivos e se já estão ou não corrigidos. Por outro lado, os erros reportados como *commit* foram considerados erros reais e classificados como corrigidos. As versões onde foram identificados erros foram selecionadas e analisadas por todas as ferramentas. Uma vez que as ferramentas de análise estática devolvem falsos positivos, seguiu-se a fase de validação dos seus relatórios de resultados. Esta validação, foi feita em duas etapas: i) comparação com o histórico de erros dos repositórios; ii) revisão manual. A comparação com o histórico de erros para além de permitir validar resultados, também permitiu calcular a percentagem de erros identificados corretamente e detetar a existência de novos erros. Os novos erros identificados foram, posteriormente, verificados através da revisão manual do código. Por fim, com as informações recolhidas foi possível identificar padrões de erros reconhecidos pelas ferramentas e construir exemplos mínimos para cada um deles. Cada uma das etapas referidas é descrita rigorosamente nas próximas secções deste capítulo.

3.1 Escolha de ferramentas

A escolha das ferramentas de análise estática foi feita com base nos seguintes critérios: i) analisarem programas em C/C++; ii) serem ferramentas de código aberto; iii) serem ferramentas de projetos ativos; iv) identificarem pelo menos dois dos seguintes erros: desreferência inválida, operação de libertação inválida, fuga de memória e acesso a variáveis não inicializadas.

Como já foi referido, as linguagens C/C++ são as mais utilizadas no desenvolvimento de projetos críticos. Além disso, os programas desenvolvidos nestas linguagens sofrem, tipicamente, erros de memória. Por estes motivos, decidiu-se analisar programas em C/C++. O primeiro critério de seleção das ferramentas foi a capacidade de análise destas linguagens.

Sendo bem conhecido o grande potencial de ferramentas comerciais como o Coverity [38], CodeSonar [12] e o PVS-Studio [56], considerou-se que seria relevante analisar o desempenho de ferramentas não comerciais. Assim, todas as ferramentas selecionadas são de código aberto, pois têm a vantagem de serem de utilização livre e acessíveis a todos.

Considerou-se também como critério de seleção as ferramentas pertencentes a um projeto ativo. É essencial que exista alguém a trabalhar atualmente na manutenção e/ou desenvolvimento da ferramenta, pois, qualquer dúvida ou defeito identificado durante a sua utilização poderá ser reportado.

O último critério foi a capacidade de identificação de pelo menos dois dos erros de memória listados anteriormente, pois podem causar o comportamento indefinido do programa.

Numa primeira seleção foram identificadas 32 ferramentas (Tabela 3.1) capazes de analisar código C/C++. Seguidamente, foram classificadas de acordo com a sua atividade e se são ferramentas de código aberto. Cerca de metade destas ferramentas foram classificadas como sendo de código aberto e a outra metade como comerciais. Das ferramentas de código aberto foram apenas selecionadas as que fazem parte de projetos ativos. Após

esta classificação restaram apenas 10 ferramentas. Posteriormente, estas foram classificadas relativamente às suas funcionalidades de verificação de projeto (Tabela 3.2), tendo sido selecionadas as ferramentas que garantissem pelo menos duas das funcionalidades listadas.

Tabela 3.1: Ferramentas de análise estática.

Ferramenta	Código aberto	Ativo
AdLint [2]	✓	x
Astrée [18]	x	✓
Axivion Bauhaus Suite [58]	x	✓
BLAST [64]	✓	x
Cppcheck [39]	✓	✓
Cppdepend [16]	x	✓
Cpplint [17]	✓	✓
Clang Static Analyzer (CSA) [11]	✓	✓
Cobra [36]	✓	✓
Codacy [4]	x	✓
CodeSonar [12]	x	✓
Coverity [38]	x	✓
ECLAIR [5]	x	✓
Flawfinder [75]	✓	✓
Fluctuat [31]	x	✓
Frama-C [13]	✓	✓
Goanna [25]	x	✓
Infer [10]	✓	✓
Klocwork Static Code Analysis [8]	x	✓
LCLint [23]	✓	x
LDRA Testbed [33]	x	✓
Parasoft C/C++ test [51]	x	✓
PC-Lint [28]	x	✓
Polyspace [77]	x	✓
Predator [19]	✓	✓
PVS-Studio [56]	x	✓
PRQA QAC [55]	x	✓
Saturno [78]	✓	x
SLAM project [7]	x	x
Splint [22]	✓	x
Uno [35]	✓	✓
VisualCodeGrepper [74]	✓	✓

Foram identificadas 6 ferramentas que cumpriam com todos os critérios referidos (cerca de 19% da lista de ferramentas inicial), que foram dispostas na Tabela 3.3, onde se fez a sua caracterização relativamente à facilidade de instalação, facilidade de utilização, teoria e correção. Nesta última tabela podemos observar que as ferramentas se baseiam em diferentes teorias para realizar a sua análise, sendo que a descrição de cada uma delas pode ser consultada na Secção 2.2 deste documento. Além disso, na Tabela 3.3, é

Tabela 3.2: Funcionalidades das ferramentas de análise estática.

Ferramenta	Fuga de memória	Desreferências inválidas	Operações inválidas de libertação de memória	Acesso a variáveis não inicializadas
Cppcheck	✓	✓	✓	✓
CppLint	x	x	x	x
Clang Static An.	✓	✓	✓	✓
Cobra	x	x	x	x
Flawfinder	x	x	x	x
Frama-C	x	✓	x	✓
Infer	✓	✓	✓	x
Predator	✓	✓	✓	x
Uno	x	✓	x	✓
VisualCodeGrepper	x	x	x	x

importante ressaltar que a correção de uma ferramenta é influenciada pelo facto de esta apresentar, ou não, filtragem de falsos negativos. Relativamente à facilidade de instalação, a maioria das ferramentas tem de ser instalada manualmente. Por outro lado, todas as ferramentas estão disponíveis nas três plataformas mais comuns (Linux, macOS e Windows) exceto o Predator, que apenas funciona em Linux. Relativamente à facilidade de utilização, metade das ferramentas selecionadas não precisa de uma função `main` e apenas o Predator requer anotações adicionais no código fonte, que apenas são necessárias para imprimir informação e não para a execução da análise. Por fim, todas as ferramentas permitem a análise de ficheiros isolados e algumas delas (Cppcheck e Frama-C) disponibilizam interface gráfica, o que facilita a sua utilização.

Tabela 3.3: Características das ferramentas de análise estática.

Ferramenta	Teoria	Correção	Instalação		Utilização			
			Fornece executável	Multi-plataforma	Requer `main`	Requer anotações	Análise de ficheiros	Interface gráfica
Cppcheck	AST	x	✓	✓	x	x	✓	✓
Clang Static Analyzer	CFG	✓	✓	✓	x	x	✓	x
Frama-C	AST	✓	x	✓	✓	x	✓	✓
Infer	SL, AI	✓	x	✓	x	x	✓	x
Predator	SMG	✓	x	x	✓	✓	✓	x
Uno	CFG	x	x	✓	✓	x	✓	x

Concluindo, no final do processo de seleção obtiveram-se as seguintes ferramentas: i) Cppcheck [39], ii) CSA[1], iii) Frama-C [13], iv) Infer [10], v) Predator [19] e vi) UNO [35]. Numa primeira análise experimental, não conseguimos que as ferramentas UNO e Frama-C produzissem resultados relevantes, pelo que optámos por excluir estas ferramentas da análise comparativa final. Nas secções seguintes é feita uma descrição breve das 4 ferramentas selecionadas.

[1] http://clang.llvm.org/

3.1.1 Cppcheck

O Cppcheck [39] é uma ferramenta de análise estática para código C/C++ que utiliza a abordagem AST para identificar comportamento indefinido e construções de código perigosas. O Cppcheck tem como objetivo detetar apenas erros reais no código, portanto, esta ferramenta devolve uma percentagem muito baixa de falsos positivos. Ou seja, o Cppcheck raramente reporta um erro que não seja real, no entanto, existem alguns erros que esta ferramenta não deteta [39]. Esta ferramenta disponibiliza uma interface gráfica, mas também pode ser utilizada diretamente através da linha de comandos.

3.1.2 Clang Static Analyzer

O Clang Static Analyzer [11] é uma ferramenta de análise estática baseada em CFGs, que suporta código C, C++ e Objective-C. Esta ferramenta é facilmente instalada em sistemas iOS, pois fornece um executável, mas também pode ser construída manualmente através do código fonte para sistemas Linux e Windows. A ferramenta fornece um relatório de erros e constrói uma versão anotada do código, na qual é possível seguir o fluxo dos dados.

O Clang Static Analyzer faz parte do projeto Clang [1], cujo objetivo é criar um novo *front-end* de um compilador LLVM [69] para linguagens de programação C, constituindo um substituído de código aberto ao GCC. Assim, o mecanismo de análise estática utilizado pelo Clang Static Analyzer é uma biblioteca Clang e tem a capacidade de ser reutilizado em diferentes contextos e por diferentes clientes [11].

3.1.3 Infer

O Infer [10], também chamado de "Facebook Infer", é uma ferramenta de análise estática baseada em SL e AI. Esta ferramenta suporta Java, Objective-C, C++ e C e está atualmente a ser usada para verificar o código das principais aplicações do Facebook para Android e iOS.

A primeira versão do Infer foi desenvolvida pela *startup* Monoidics[2], cujos fundadores participaram no desenvolvimento da SL. A SL é uma teoria que permite que a análise do Infer se baseie em partes pequenas e independentes da memória da aplicação que está a verificar. A partir desta teoria foram desenvolvidos vários protótipos de ferramentas e técnicas de inferência subjacentes, que resultaram na descoberta da bi-abducção como forma de analisar programas de forma modular. A técnica de bi-abducção é utilizada pelo Infer como forma de descobrir propriedades sobre o comportamento de partes independentes do código. Ao armazenar essas propriedades entre as execuções, o Infer precisa apenas de analisar as partes do projeto que mudaram, reutilizando os resultados da análise anterior. Em 2009, como resultado do sucesso destas pesquisas, foi fundada a Monoidics, que viria a ser adquirida pelo Facebook em 2013 [50].

[2] https://www.crunchbase.com/organization/monoidics

O processo de análise do Infer é dividida em duas fases, independentemente da linguagem do programa a analisar [10]:

Fase de captura: o Infer vai traduzir os ficheiros que queremos analisar numa linguagem intermédia. Esta tradução é similar à compilação, portanto, o Infer pode usar alguma da informação do processo de compilação para executar a sua própria tradução.

Fase de análise: os resultados da tradução vão ser analisados pelo Infer. A análise é feita função a função, sempre que o Infer encontrar um erro vai reportar esse erro e passar para a função seguinte. Os erros encontrados são filtrados e são apenas apresentados na consola aqueles que forem considerados reais. No entanto, os restantes erros ficam registados num ficheiro, que pode ser consultado.

3.1.4 Predator

O Predator [19] é uma ferramenta de análise estática de código sequencial C que funciona como *plug-in* do GCC. Esta ferramenta é baseada em SMGs e foi estendida para suportar várias formas de manipulação de memória de baixo nível. Tais operações incluem aritmética de ponteiros, uso seguro de ponteiros inválidos, reinterpretação do conteúdo da memória, etc. O Predator é capaz de detetar ou provar ausência de vários tipos de erros de memória [19]:

- Fuga de memória;
- Desreferências inválidas;
- Operações de libertação de memória inválidas;

As verificações de fuga de memória são feitas recolhendo os endereços perdidos em cada operação do programa e, seguidamente, verificando se a acessibilidade dos objetos alocados depende dos endereços recolhidos.

3.2 Escolha de projetos a analisar

Os projetos foram selecionados do GitHub e os critérios utilizados na sua seleção foram os seguintes: i) serem código aberto; ii) terem uma dimensão variada, dentro do intervalo de 50 a 128000 kB; iii) serem escritos em C/C++; iv) terem histórico de erros (e.g., *commits*); v) terem elevada popularidade no GitHub. Seguindo os critérios listados foram selecionados inicialmente 16 projetos que foram, posteriormente, classificados e ordenados de acordo com os seguintes critérios:

Prioridade: a nossa avaliação ponderada (1 = mais prioritário) dos demais critérios, tendo em especial consideração a quantidade de operações de manipulação de memória (`malloc`, `calloc`, `realloc`, `free` e utilização de ponteiros) e tipo de impacto que os erros têm.

Tabela 3.4: Classificação das dimensões dos Programas.

Classificação	Dimensão	
	(kB)	(# linhas)
Pequeno	50–400	2000–6000
Médio	400–1600	6000–64000
Grande	16000–128000	64000–512000

Popularidade: determinada pela quantidade de estrelas atribuídas pelos utilizadores ao repositório no GitHub.

Número de versões: favorecemos os programas com múltiplas versões, verificando se uma ferramenta identifica os erros presentes numa determinada versão e confirma que os mesmos foram corrigidos nas versões seguintes.

Dimensão: assumindo que 1kB corresponde aproximadamente a 40 linhas no programa fonte, os programas foram divididos por dimensão (Tabela 3.4). Agrupámos programas com a mesma dimensão para facilitar a seleção, uma vez que nos interessava testar projetos com dimensões diferentes.

Erros de memória: número e percentagem de erros de memória reportados nos *commits*.

Falha: impacto dos erros identificados no projeto, classificado em 3 categorias (por ordem decrescente): falha, performance e segurança. Consideramos mais prioritários programas com maior percentagem de falhas.

Questões sobre erros: submetidas pelos utilizadores e referentes a erros de memória identificados durante a instalação ou utilização do projeto. Em alguns casos, os programas poderão não ter erros de memória identificados, mas ter uma grande quantidade de questões sobre esse tipo de erros por responder ou resolver.

Manipulação de memória: operações de manipulação de memória e quantidade de ponteiros utilizados.

Seguindo os critérios apresentados, os programas foram agrupados por dimensão, e dentro de cada um desses grupos foi feita uma ordenação por prioridade. Os projetos com a mesma dimensão e prioridade foram ordenados de acordo com a sua popularidade. Na Tabela 3.5 estão listados, de forma já ordenada, todos os programas analisados e respetivas características. Sendo que, os 4 projetos escolhidos (um de pequena dimensão, um de média dimensão e dois de grande dimensão) são os que se encontram posicionados no topo de cada um grupos de dimensões. Desta forma, os projetos escolhidos para esta experiência têm diferentes dimensões, prioridade igual a 1 e elevada popularidade.

O SQLite tem 188000 kB de dimensão, sendo, portanto, superior ao intervalo estabelecido anteriormente para os projeto de grande dimensão (Tabela 3.4), por este motivo a dimensão deste projeto foi classificada como "muito grande". Este projeto possui um

Tabela 3.5: Características dos programas.

projeto	Prioridade	Dimensão	Popularidade (# Estrelas)	Versões	Erros de memória (%) total erros	Erros causam falha (%) erros mem.	Casos (*issues*) sobre erros de memória (%) total casos	Manipulação de memória # Operações	Manipulação de memória # Apontadores
SDS	1	Pequena	2215	2	0	0	23	121	71
TinyVM	1	Pequena	1254	1	25	4	25	64	115
Twemcache	1	Pequena	811	8	0	0	21	257	205
GloVe	2	Pequena	2204	2	18	0	28	118	64
Sparkey	2	Pequena	784	2	5	0	0	86	69
Beanstalkd	1	Média	4405	34	6	7	10	93	129
Openwebrtc	1	Média	1519	1	3	1	2	439	323
Redcarpet	2	Média	4169	23	4	0	7	66	44
Http-Parser	2	Média	3816	20	5	0	5	30	96
Leveldb	3	Média	12915	18	4	6	7	37	173
Tmux	1	Grande	9530	21	8	11	3	1116	918
SQLite	1	Muito Grande	-	100	5	3	-	6650	6096
Memcached	1	Grande	7452	73	3	7	9	591	395
The foundation	1	Grande	2810	670	3	6	7	682	1342
Timescaledb	1	Grande	4259	25	3	1	3	59	214
Lwan	2	Grande	4131	1	15	8	21	462	700
Ziparchive	2	Grande	3368	37	1	2	3	67	284

repositório não oficial no GitHub de onde foi possível retirar informação sobre o tipo de erros e impacto que estes provocam no projeto. Este repositório, apesar de não ser o oficial é atualizado de acordo com o mesmo e disponibiliza as 100 versões já lançadas deste projeto. A única informação que não se encontra disponível no repositório do GitHub são os *issues*. Além disso, a quantidade de estrelas atribuídas a este repositório não corresponde à verdadeira popularidade do SQLite. Assim, após análise do repositório obteve-se que cerca de 31% dos seus *commits* são relativos a erros, sendo que 5% são erros de memória. Relativamente ao impacto dos erros no projeto, cerca de 3% causam a falha do programa e 8% afetam a sua performance. Por último, existem mais de 6500 operações de manipulação de memória e cerca de 6000 apontadores no código fonte do SQLite.

Concluindo, os projetos selecionados para análise foram os seguintes:

Simple Dynamic String (SDS):[3] biblioteca de *strings* dinâmicas, projetada para aumentar as funcionalidades limitadas das *strings* da biblioteca C;

Beanstalkd:[4] gestor de tarefas para aplicações distribuídas;

Tmux:[5] multiplexador de terminal, que permite que uma série de terminais possam ser acedidos e controlados a partir de um único terminal;

SQLite:[6] biblioteca C que implementa um mecanismo de base de dados SQL transacional autónoma, sem servidor e com configuração zero.

[3] https://github.com/antirez/sds
[4] https://github.com/kr/beanstalkd
[5] https://github.com/tmux/tmux
[6] https://www.sqlite.org

3.3 Escolha das versões dos projetos

Após serem selecionados os projetos a analisar, o repositório de cada um deles foi estudado com o objetivo de obter informações sobre as versões onde foram identificados e corrigidos erros de memória. Este estudo foi feito por três motivos: i) verificar se as ferramentas identificam corretamente quando um erro foi corrigido entre versões. ii) os projetos de maior dimensões têm demasiadas versões disponíveis. iii) nem todas as versões têm alterações relevantes. Assim, foram selecionadas 2 versões para o SDS, 12 versões para o Beanstalkd, 15 versões para o Tmux e 5 versões para o SQLite. A escolha de apenas 5 versões para o SQLite deve-se ao facto de termos considerado que não seria relevante analisar as versões mais antigas deste projeto, uma vez que este possui 100 versões lançadas desde 2000. Portanto, optou-se por analisar apenas as versões lançadas no último ano.

3.4 Validação de resultados

Como referido na Secção 2.2, as ferramentas de análise estática fazem sobre-aproximação como forma de tomar decisões sobre propriedades indecidíveis do código. Uma consequência da sobre-aproximação é a identificação de falsos positivos. Portanto, foi necessário validar os resultados devolvidos pelas ferramentas. A validação de resultados foi realizada em duas fases:

Comparação com o histórico de erros: os dados do histórico de erros dos repositórios, recolhidos na fase de escolha de versões dos projetos (Secção 3.3), foram utilizados para validar os relatórios de resultados das ferramentas através da sua comparação com os mesmos.

Revisão manual de código: os novos erros, que não fazem parte do histórico de erros do repositório, foram sujeitos a revisão manual de código, de forma a verificar se eram erros reais ou falsos positivos.

Após feita a validação dos resultados, os novos erros classificados como reais foram reportados nos respetivos repositórios.

Capítulo 4

Experimentação e Análise

No Capítulo 3 foram descritas as várias fases da metodologia de trabalho seguida durante a elaboração desta dissertação. Na primeira fase foram selecionadas as ferramentas a utilizar neste estudo: i) Cppcheck; ii) Clang Static Analyzer (CSA); iii) Infer; iv) Predator. Seguidamente, na fase de escolha de projetos, foram selecionados 4 softwares considerados relevantes e com diferentes dimensões: i) Simple Dynamic String (SDS), com dimensão pequena; ii) Beanstalkd, dimensão média; iii) Tmux, dimensão grande; iv) SQLite, dimensão muito grande. Por fim, foram selecionadas as versões dos projetos a analisar: 2 versões no SDS, 12 versões no Beanstalkd, 15 versões no Tmux e 4 versões no SQLite. Neste capítulo serão apresentados e analisados os relatórios de resultados obtidos pelas ferramentas aplicadas às várias versões dos projetos selecionados.

4.1 Relatório de resultados

A seleção das ferramentas de análise estática seguiu um conjunto de critérios (Secção 3.1). Um desses critérios foi a capacidade de identificação de erros de memória, sendo que, inicialmente, foram considerados apenas 4 tipos de erros: fuga de memória, desreferência inválida, operação de libertação de memória inválida e acesso a variáveis não inicializadas. No entanto, as ferramentas selecionadas são capazes de identificar outros tipos de erros de memória, pelo que, após a sua execução foram registados os dados relativos a estes erros. Assim, na Tabela 4.1 estão listados todos os tipos erros de memória identificados pelas ferramentas nos vários casos de estudo. Nesta tabela podem ser consultados os acrónimos utilizados em todas as tabelas e gráficos no restante documento e uma breve descrição dos erros.

Os resultados apresentados nesta secção dizem respeito aos softwares SDS, Beanstalkd, Tmux e SQLite. O SDS é um projeto de pequena dimensão com apenas 2 versões no seu

Tabela 4.1: Diferentes tipos de erros reportados pelas ferramentas.

Nome	**Acrónimo**	**Descrição**
Fuga de memória	ML	A memória alocada pelo programa não é libertada quando deixa de ser necessária.
Fuga de recursos	RL	Os recursos utilizados pelo programa não são libertados quando deixam de ser necessários.
Desreferência inválida	InvD	Uma desreferência é considerada inválida se ocorrer uma das três circunstâncias seguintes: i) Desreferência de um apontador com o valor NULL (ND); ii) Desreferência de um apontador que já foi libertado ou não foi alocado (UAF); iii) Desreferência de um apontador fora dos limites (OB).
Operação inválida de libertação de memória	InvF	Uma operação de libertação de memória é considerada inválida se ocorrer uma das seguintes situações: i) A memória não foi alocada anteriormente; ii) A memória já foi libertada; iii) A memória, que queremos libertar, encontra-se a meio de um bloco alocado.
Acesso a variáveis não inicializadas	UV	Uma variável não inicializada é uma variável que é declarada, mas não lhe é atribuído nenhum valor conhecido antes de ser usada.
Dead store	DS	Ocorre quando é atribuído um valor a uma variável, mas este nunca é utilizado.
Falha de segmentação	SegF	Ocorre quando um programa tenta aceder a um endereço de memória que não existe ou que está reservado para outro programa.
Apontador pendente	DP	Um apontador que não referência um endereço válido.
Sobrecarga do *buffer*	BO	Ocorre quando um programa, ao escrever dados num *buffer*, ultrapassa os seus limites e sobrescreve a memória adjacente.

repositório. Além disso, não existem erros de memória reportados como *commits* no repositório deste software, mas existem vários erros deste tipo identificados por utilizadores (i.e., *issues*), sendo que alguns desses erros ainda se encontram por corrigir e/ou validar. Portanto, todos os erros reportados no repositório e devolvidos pelas ferramentas foram verificados através de revisão manual de código. Na Tabela A.1 (Apêndice A, página 79) estão representados os erros identificados nas 2 versões do SDS pelas 4 ferramentas.

O Beanstalkd é um projeto de dimensão média e possui 34 versões no repositório. As ferramentas foram executadas em 12 versões deste software onde haviam sido identificados erros de memória, permitindo assim comparar os resultados destas com o histórico de erros disponível. Na Tabela A.2 (Apêndice A, página 75) estão representados os erros identificados nas 12 versões do Beanstalkd. De notar que, na tabela referida foram omitidos 129 falsos positivos identificados pelo Predator. Na última versão deste software foram

identificados 5 erros novos, que ainda não se encontravam reportados no repositório.

O Tmux é um projeto de dimensão grande e possui 21 versões no repositório. Após análise do histórico de erros foram selecionadas 15 versões onde foram executadas as ferramentas. Na Tabela A.3 (Apêndice A, página 76) estão listados os erros identificados na última versão do Tmux (versão 2.7). Note-se que, apesar dos erros terem sido reportados na última versão deste software, a sua maioria foi introduzida em versões mais antigas, tendo persistido no código.

Por fim, o SQLite é um projeto de dimensão muito grande e possui 100 versões, lançadas desde 2000. Tomou-se a decisão de analisar apenas as versões lançadas no último ano, o que resultou na execução das ferramentas em 4 versões deste software. Na Tabela A.4 (Apêndice A, página 79) estão listados os erros identificados na última versão do SQLite (versão 3.24.0). Na secção seguinte é feita uma descrição da análise dos resultados obtidos para cada um dos softwares.

4.2 Análise dos resultados

Os resultados obtidos pela execução das ferramentas de análise estática foram comparados com o histórico de erros dos repositório e os novos erros foram sujeitos a revisão manual de código. Assim, a partir da análise dos resultados foi possível calcular a percentagem de falsos positivos (FP), falsos negativos (FN), verdadeiros positivos (TP) e verdadeiros negativos (TN) identificados por cada uma das ferramentas. Para uma determinada ferramenta a classificação dos erros em cada uma das categorias segue os seguintes critérios:

Falso positivo: erro não real mas reportado pela ferramenta.

Falso negativo: erro real mas não reportado pela ferramenta.

Verdadeiro positivo: erro real e reportado pela ferramenta.

Verdadeiro negativo: erro não real e não reportado pela ferramenta.

Um erro é considerado real quando está reportado no histórico de erros do repositório ou quando é identificado por uma das ferramentas e validado com revisão manual de código.

Clarificando, um verdadeiro negativo é um erro reportado por pelo menos uma ferramenta, mas que, após revisão manual de código, se verificou ser um falso positivo. Este erro será, para as ferramentas que não o identificaram, um verdadeiro negativo.

Uma ferramenta será tão útil quanto menor for o número de falsos positivos e falsos negativos que devolve [68]. Portanto, com base nos resultados obtidos para a distribuição de erros, foram calculadas as seguintes métricas:

Eficácia: percentagem de verdadeiros positivos que uma ferramenta reporta do total de erros reais existentes no projeto. A formula da eficácia é a seguinte:

$$e = \frac{tp}{tp + fn} \tag{4.1}$$

Precisão: percentagem de verdadeiros positivos que uma ferramenta reporta. A formula da precisão é a seguinte:

$$p = \frac{tp}{tp + fp} \tag{4.2}$$

De notar que todos os testes foram realizados na mesma máquina Linux, cujo sistema operativo é o Ubuntu 16.04. Desta forma foi possível fazer uma comparação válida dos resultados e do tempo de execução das ferramentas.

4.2.1 SDS

Na versão 1 foram reportados 6 erros no repositório, sendo um desses um falso positivo identificado por um utilizador. Na versão 2 foram reportados apenas 2 erros no repositório, mas um já existia na versão 1. No total foram registados 28 erros, dos quais 21 são novos erros identificados pelas ferramentas, que não fazem parte do histórico de erros do repositório. No entanto, apenas 6 dos 21 erros poderão ser classificados como reais e 2 desses erros foram identificados na última versão do projeto. Assim, os erros reais que identificámos e reportámos na última versão do SDS (versão 2.0.0) são:

- *sds.c:1123: error: null dereference*, identificado pelo Infer[1];
- *sds.c:1240: error: dead store*, identificado pelo CSA e pelo Infer[2].

4.2.1.1 Tempo de permanência dos erros

Na Tabela 4.3 estão listados os tempos de permanência de cada verdadeiro positivo identificado no SDS por pelo menos uma ferramenta. Nesta tabela estão incluídos os 2 novos erros, listados na Secção 4.2.1, e 2 erros que foram identificados pelas ferramentas, mas já faziam parte do histórico de erros do repositório. O primeiro erro foi identificado pelo Cppcheck na versão 1 e consiste numa fuga de memória. O Infer identificou este erro na linha seguinte à do Cppcheck, no entanto, esta ferramenta reportou-o como sendo uma desreferência nula. Este erro permaneceu cerca de 9 meses no software até ser identificado e corrigido. O segundo erro foi identificado apenas pelo Infer na versão 2, consiste uma desreferência nula e permaneceu cerca de 30 meses no software. Os restantes erros listados na Tabela 4.3 permaneceram entre 17 a 30 meses no código fonte do SDS.

Como referido anteriormente, os erros reportados no repositório deste software foram encontrados por utilizadores, ou seja, após o seu lançamento. A utilização das ferramentas

[1] https://github.com/antirez/sds/issues/99
[2] https://github.com/antirez/sds/issues/100

Tabela 4.3: Datas e tempo em meses decorrido desde a introdução até à correção dos erros no SDS.

Erro		Introduzido	Corrigido	Intervalo de tempo
Localização	Tipo			(meses)
sds.c:159	ML	06/02/2014	25/11/2014	9
sds.c:160	ND	06/02/2014	25/11/2014	9
sds.c:891	ND	06/02/2014	25/07/2015	17
sds.c:894	ND	06/02/2014	25/07/2015	17
sds.c:1123	ND	06/02/2014	x	x
sds.c:92	ND	25/07/2015	07/02/2018	30
sds.c:1240	DS	25/07/2015	x	x

de análise estática durante a fase de desenvolvimento do SDS teria permitido corrigir estes erros mais cedo.

4.2.1.2 Distribuição dos erros

Na Figura 4.1 está representado um gráfico com as percentagens de falsos positivos, verdadeiros positivos, falsos negativos e verdadeiros negativos identificados pelas ferramentas no SDS.

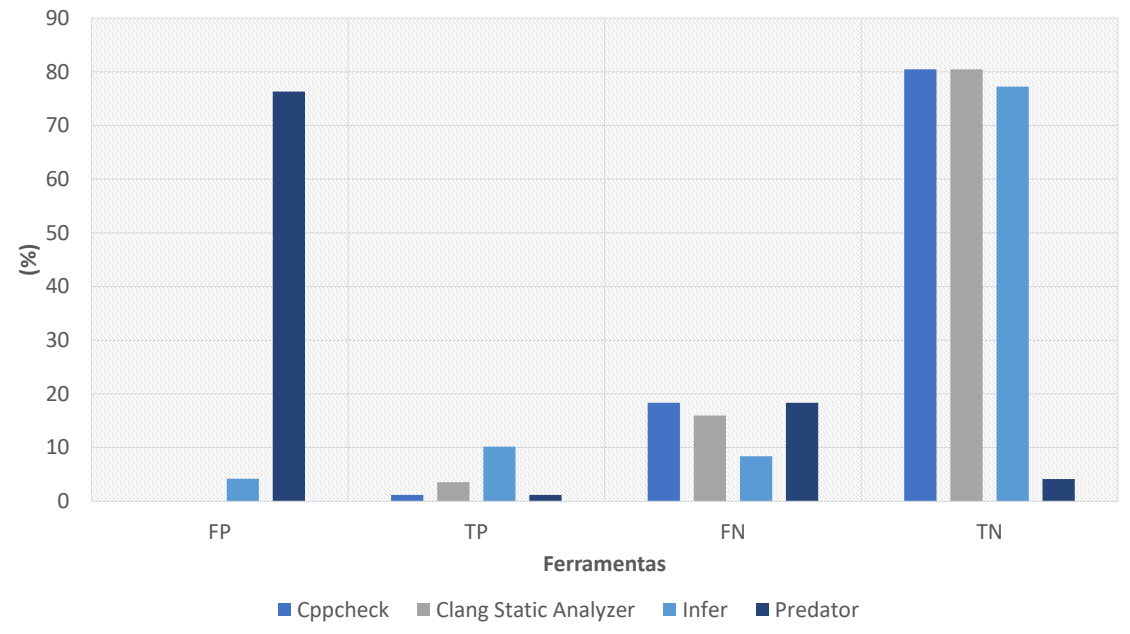


Figura 4.1: Distribuição dos vários tipos de erros identificados no SDS pelas ferramentas.

Este software retorna, intencionalmente, apontadores para o meio de blocos de memória alocados com `malloc`. Portanto, as operações de desreferência ou libertação de memória aplicadas a ponteiros nestas situações são reportadas pelas ferramentas CSA e Predator como erros, que nós classificámos como falsos positivos. Apesar disto, verificámos que o CSA apresentou uma percentagem relativamente baixa de falsos positivos. Por outro lado, verificou-se 30% de falsos negativos. Esta percentagem deve-se ao facto da CSA não identificar um padrão de erros relevante, nomeadamente a verificação dos resultados de operações de alocação de memória (i.e., `malloc`, `calloc` e `realloc`).

Por princípio, a ferramenta Predator não permite chamadas de funções externas, a fim de excluir qualquer efeito colateral que possa potencialmente quebrar a segurança da memória. As únicas funções externas permitidas são aquelas que o Predator reconhece como funções integradas e as modela apropriadamente, provando a segurança da memória (`malloc`, `free`, e algumas funções da biblioteca do C tais como `memset`, `memcpy` e `memmove` [19]). Assim, os erros reportados por esta ferramenta são na sua maioria falsos positivos, resultantes de esta ignorar chamadas a funções externas não modeladas. Ou seja, para que a ferramenta seja eficiente o programa verificado deve ser fechado, isto é, deve alocar e inicializar todas as estruturas de dados usadas. Por este motivo, o Predator não foi capaz de identificar nenhum erro real neste software, sendo a ferramenta que devolveu a maior percentagem de falsos positivos e falsos negativos.

O Infer é a ferramenta que devolve uma maior percentagem de verdadeiros positivos e a menor percentagem de falsos negativos neste software. Contudo, o Infer falha na identificação de alguns erros reais, que seguem um padrão de erro reconhecido pela ferramenta. A análise realizada pelo Infer consiste numa execução simbólica do código, mantendo uma *heap* simbólica, que tenta garantir a segurança da memória do programa, quando a ferramenta não consegue provar a segurança da memória, pode reportar um erro, se encontrar uma desreferência nula ou uma fuga de memória, ou pode perceber que se encontra num estado inconsistente. Em ambos os casos, a análise é interrompida. Outra razão para que o Infer não consiga reportar erros que poderia identificar é a existência de um tempo limite para execução da análise, que é por vezes atingido antes desta chegar ao fim [10], e que parece não ser possível de parametrizar.

O Cppcheck é a ferramenta que devolve a menor quantidade de erros. Por uma opção de engenharia, esta ferramenta realiza filtragem dos erros para reduzir o número de falsos positivos reportados, e, consequentemente, acaba por eliminar erros reais gerando falsos negativos.

4.2.1.3 Eficácia e Precisão

Com base na quantidade de falsos positivos, verdadeiros positivos e falsos negativos foi possível calcular os valores de eficácia e precisão das 4 ferramentas no SDS. Estes valores estão representados no gráfico da figura Figura 4.2.

O Cppcheck tem uma precisão de 100%, no entanto tem uma eficácia bastante baixa.

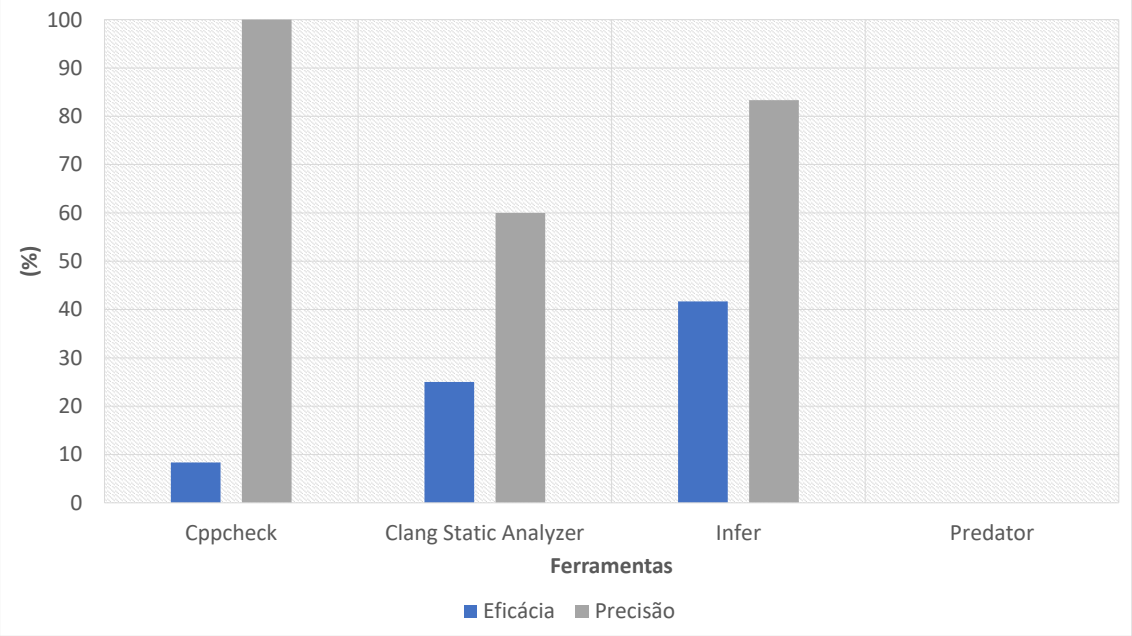


Figura 4.2: Valores de eficácia e precisão das 4 ferramentas no SDS.

Ou seja, esta ferramenta identificou poucos erros, no entanto todos os erros identificados por esta ferramenta foram todos classificados como verdadeiros positivos. O Infer e o CSA tem valores de precisão acima dos 50%, por outro lado, apresentação valores de eficácia abaixo dos 50%. O Predator, como seria de esperar, tem eficácia e precisão igual a 0%.

4.2.1.4 Tempo de execução

Na Figura 4.3 está representado o gráfico dos tempos de execução das ferramentas no SDS que, como se pode observar, é bastante baixo, não ultrapassando os 16 segundos.

A partir da análise do gráfico podemos perceber que o tempo de execução aumenta na versão 2 para todas as ferramentas, sendo que o menor e maior desvio padrão são obtidos pelo Cppcheck e pelo CSA, respetivamente. O Infer é a ferramenta com o tempo de execução mais elevado, no entanto, é também a ferramenta que devolveu a maior percentagem de verdadeiros positivos (Tabela 4.1). Da mesma forma, o Cppcheck é a ferramenta com o tempo de execução mais reduzido e que devolveu a menor quantidade de erros de qualquer tipo. O Predator apresenta, de igual forma, tempos de execução bastante baixos, pois a sua execução é interrompida nos casos em que o código faz chamadas a funções externas.

4.2.2 Beanstalkd

No Beanstalkd foram identificados manualmente 182 erros, dos quais 168 foram identificados pelas ferramentas. Desses 168 erros, apenas 4 já estavam reportados no repositório, ou seja, as ferramentas reportaram 164 novos erros. No entanto, apenas 20 desses erros foram classificados como erros reais. Na última versão do projeto foram identificados 2

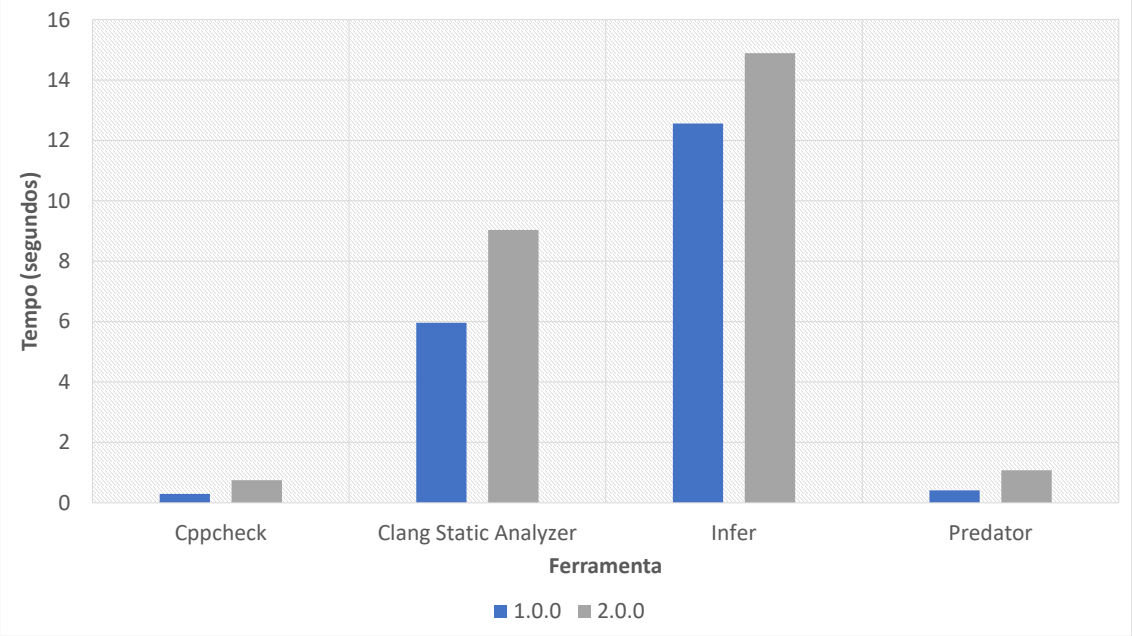


Figura 4.3: Tempo de execução das ferramentas no SDS.

novos erros considerados únicos, ou seja, com origens diferentes. Assim, os novos erros identificados e reportados no repositório do Beanstalkd foram os seguintes:

- *prot.c:501: error: null dereference*, identificado pelo Infer[3];
- *testheap.c:222: error: memory leak*, identificado pelo Cppcheck e Predator[4].

Estes erros, tal como aconteceu para o software SDS, foram reportados e aguardam *feedback* por parte dos programadores responsáveis pelo repositório.

4.2.2.1 Tempo de permanência dos erros

Na Tabela 4.4 estão listados os tempos de permanência de cada verdadeiro positivo identificado nas várias versões do Beanstalkd por pelo menos uma ferramenta. Nesta tabela estão incluindo os novos erros identificados, que ainda se encontram por corrigir, e os 4 erros reportados no repositório e, simultaneamente, identificados pelas ferramentas. O primeiro desses erros consiste numa fuga de memória e foi identificado na versão 1.4.2 por um utilizador com o auxilio do Valgrind[5]. Este erro foi também identificado pelo Predator, uma das ferramentas em estudo, e permaneceu no software durante um mês. O segundo e terceiro erros, ambos desreferências nulas, foram identificados na versão 1.6 graças à análise realizada com o Coverity Scan[6], sendo que também foi identificado pelo

[3] https://github.com/kr/beanstalkd/issues/384
[4] https://github.com/kr/beanstalkd/issues/382
[5] http://valgrind.org/
[6] https://scan.coverity.com/

CSA. Estes erros permaneceram no software durante 3 meses. O quarto erro, que consiste num *dead store*, foi identificado na versão 1.9 pelo Infer e Predator e permaneceu no software durante 15 meses. Portanto, antes do presente estudo já tinham sido utilizadas ferramentas automáticas na verificação e validação do Beanstalkd. No caso do Coverity Scan, a sua análise foi realizada até à versão 1.9 do software e foram identificados 11 erros, dos quais foram corrigidos 8 [15].

Tabela 4.4: Datas e tempo em meses decorrido desde a introdução até à correção dos erros no Beanstalkd.

Erro		**Introduzido**	**Corrigido**	**Intervalo de tempo**
Localização	Tipo			(meses)
net.c:28	DS	08/11/2007	03/10/2009	22
beanstalkd.c:41	RS	08/11/2007	28/01/2012	50
reserve.c:51	ND	08/11/2007	28/01/2012	50
prot.c:140	ND	08/11/2007	28/01/2012	50
beanstalkd.c:395	ND	08/11/2007	17/01/2009	14
beanstalkd.c:737	ML	08/11/2007	11/12/2007	1
prot.c:320	ND	01/02/2008	28/01/2012	47
prot.c:374	ND	01/02/2008	28/01/2012	47
prot.c:1583	ND	17/01/2009	x	x
cut.c:222	RL	17/02/2009	28/01/2012	36
binlog.c:215	ND	14/10/2009	18/10/2009	0,13
binlog.c:723	ND	14/10/2009	28/01/2012	27
job.c:70	ML	18/10/2009	30/11/2009	1
net.c:31	DS	28/01/2012	14/04/2013	14
prot.c:514	ND	28/01/2012	03/11/2012	9
prot.c:554	ND	28/01/2012	03/11/2012	9
walg.c:426	RL	28/01/2012	x	x
file.c:204	ND	10/05/2012	02/09/2012	3
file.c:325	ND	10/05/2012	02/09/2012	3
prot.c:501	ND	10/05/2012	x	x
darwin.c:84	DS	03/11/2012	x	x
walg.c:416	DS	13/04/2013	05/08/2014	15
testheap.c:222	ML	05/08/2014	x	x

4.2.2.2 Distribuição dos erros

Na Figura 4.4 está representado o gráfico com as percentagens de tipos de erros identificados pelas ferramentas no Beanstalkd.

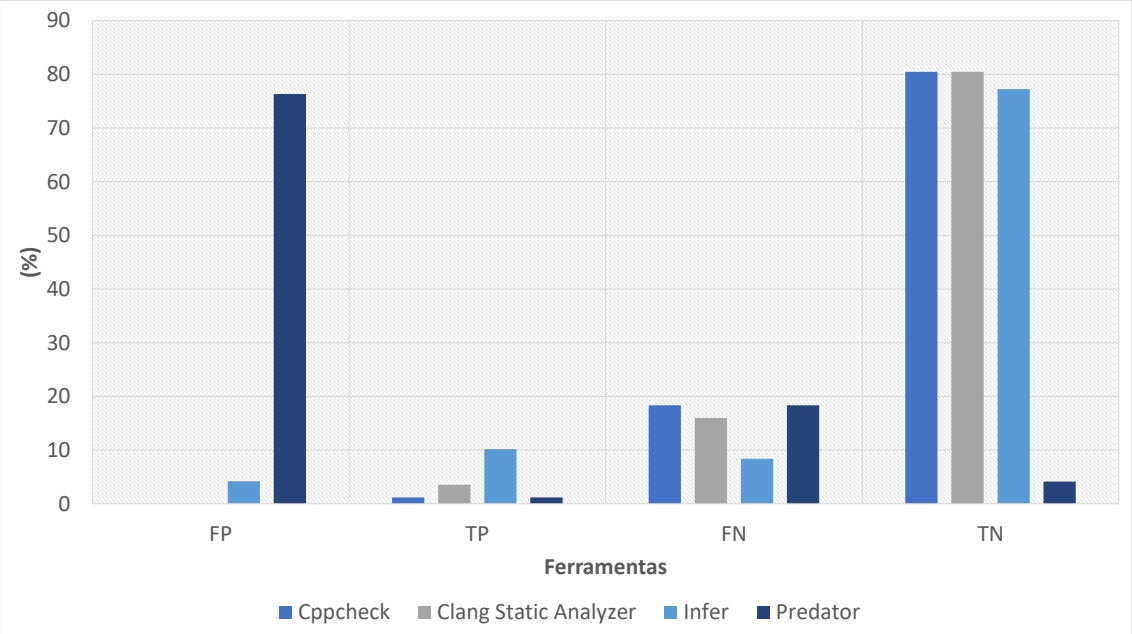


Figura 4.4: Distribuição dos vários tipos de erros identificados no Beanstalkd pelas ferramentas.

O Predator não está preparado para analisar programas muito complexos [19], pelo que a análise do Beanstalkd revelou-se pouco conclusiva, uma vez que este software tem uma dimensão e complexidade maior que o SDS. Numa tentativa de extrair algum tipo de resultado relevante, o Predator foi utilizado para testar cada um dos ficheiros do software individualmente, tendo sido obtidos 129 falsos positivos (cerca de 70% do total de erros reportados), que se devem às chamadas de funções externas que são ignoradas.

O Infer foi mais uma vez a ferramenta que devolveu uma maior percentagem de verdadeiros positivos. No entanto, foi também a ferramenta com a segunda maior percentagem de falsos positivos.

O Cppcheck não devolveu falsos positivos, no entanto, revelou-se pouco eficaz devolvendo a menor percentagem de verdadeiros positivos e a maior percentagem de falsos negativos. Estes resultados devem-se ao facto desta ferramenta fazer filtragem de falsos positivos.

Por fim, o CSA foi a ferramenta que devolveu uma maior quantidade de verdadeiros negativos (80%), juntamente com o Cppcheck. Além disso, esta ferramenta identificou a segunda maior percentagem de verdadeiros positivos no Beanstalkd.

4.2.2.3 Eficácia e Precisão

Os valores de eficácia e precisão das 4 ferramentas no Beanstalkd, encontram-se representados no gráfico da Figura 4.5. O Predator apresenta valores de eficácia e precisão

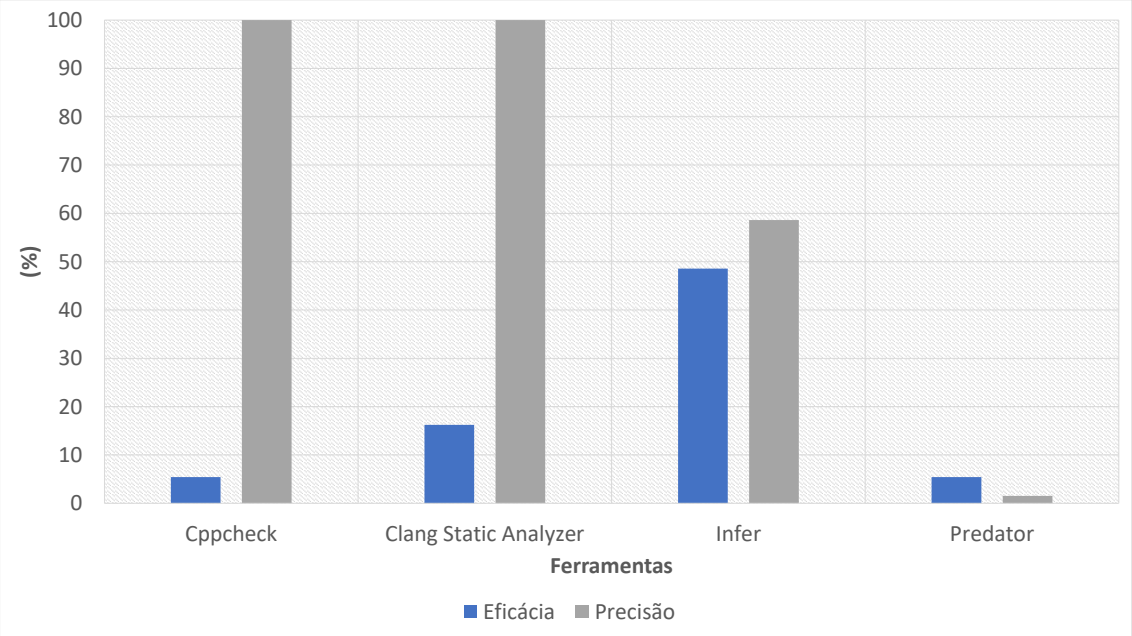


Figura 4.5: Valores de eficácia e precisão das 4 ferramentas no Beanstalkd.

diferentes de 0% neste projeto, ao contrário do que se verificou para o SDS. No entanto, esta ferramenta devolve uma grande percentagem de falsos positivos o que influencia a sua precisão. O CSA e o Cppcheck apresentam uma precisão de 100% e uma eficácia que não ultrapassa os 20%. O Infer é a ferramenta com a eficácia mais elevada (aproximadamente 50%). No caso da precisão, a ferramenta apresenta um valor de aproximadamente 60%.

4.2.2.4 Tempo de execução

Na Figura 4.6 estão representados os tempos de execução das ferramentas no Beanstalkd. As ferramentas que obtiveram os menores tempos de execução foram o Cppcheck e o Predator, tal como se tinha observado para o SDS.

Os tempos de execução obtidos para o Beanstalkd foram, naturalmente, superiores aos registados para o SDS e aumentam à medida que as versões se tornam mais recentes. A ferramenta que registou os valores de tempo de execução e desvio padrão mais elevados foi o Infer, a ferramenta com a maior percentagem de verdadeiros positivos reportados (Tabela 4.4).

O CSA, a ferramenta com a segunda maior percentagem de verdadeiros positivos reportados (Tabela 4.4), obteve valores de tempo de execução intermédios, que não ultrapassam os 20 segundos.

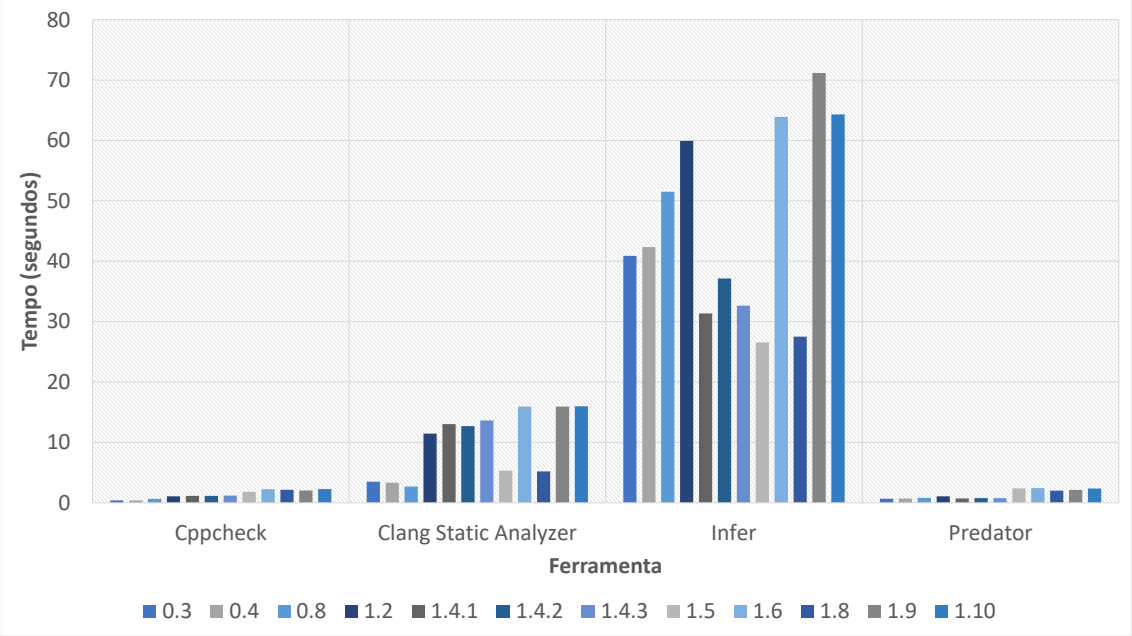


Figura 4.6: Tempo de execução das ferramentas no Beanstalkd.

O Predator, tal como havia sido verificado para o SDS, obteve tempos de execução bastante baixos, o que é justificado com a interrupção da sua execução devido às chamadas a funções externas.

O Cppcheck obteve também tempos de execução bastante baixos, similares aos obtidos para o Predator, no entanto, esta ferramenta termina a sua análise sem que ocorram interrupções.

4.2.3 Tmux

Na versão 2.7 do Tmux foram identificados 72 erros, dos quais 70 foram reportados pelas ferramentas, sendo que todos eles são novos erros. No entanto, apenas 7 desses erros foram classificados como erros reais. Os gráficos e tabelas desta secção não incluem a ferramenta Predator, pois esta não devolveu resultados conclusivos para nenhuma das versões do Tmux.

4.2.3.1 Tempo de permanência dos erros

Na Tabela 4.5 estão representados os tempos de permanência dos erros classificados como verdadeiros positivos e identificados por pelo menos uma ferramenta no Tmux. Nesta tabela estão apenas listados os erros reportados na versão 2.7 do Tmux, pelo que nenhum dos erros foi corrigido até ao momento.

Na versão 2.7 do Tmux as ferramentas não identificaram nenhum dos erros que já estavam reportados no repositório do projeto. No entanto, os resultados obtidos nas restantes versões foram comparados com o histórico de erros do Tmux e identificaram-se vários casos em que a ferramenta reportou o erro numa versão mais antiga do que a registada no

Tabela 4.5: Datas e tempo em meses decorrido desde a introdução até à correção dos erros no Tmux.

Erro		**Introduzido**	**Corrigido**	**Intervalo de tempo**
Localização	Tipo			(meses)
cmd-queue.c:189	ML	20/abr/17	x	x
cmd-respawn-window.c: 73	ND	05/nov/09	x	x
layout.c:83	UAF	22/mar/18	x	x
window.c: 668	UAF	18/out/15	x	x
key-bindings.c:108	UAF	18/out/15	x	x

repositório. Ou seja, alguns dos erros estiveram presentes em várias versões do software até serem identificados e reportados. No total verificamos que esta situação ocorreu em 5 situações distintas:

- *cmd-set-buffer.c:59: error:memory leak*, este erro foi introduzido na versão 1.1 (5 de Novembro de 2009), na qual foi reportado pelo Infer, e apenas foi identificado no repositório na versão 1.4 (27 de Dezembro de 2010) e corrigido na versão 1.5 (09 de Julho de 2011).

- *cmd-load-buffer.c:170: error:memory leak*, este erro foi introduzido na versão 1.7 (13 de Outubro de 2012), na qual foi reportado pelo Infer, e apenas foi identificado na versão 1.8 (26 de Março de 2013) e corrigido na versão 1.9a (22 de Fevereiro de 2014).

- *cmd-save-buffer.c:148: error:dead store*, este erro foi introduzido na versão 1.8 (26 de Março de 2013), na qual foi reportado pelo CSA, e apenas foi identificado na versão 1.9a (22 de Fevereiro de 2014) e corrigido na versão 2.0 (6 de Março de 2015).

- *cmd-save-buffer.c:102: error:null dereference*, este erro foi introduzido na versão 1.9a (22 de Fevereiro de 2014), na qual foi reportado pelo CSA, e apenas foi identificado na versão 2.3 (29 de Setembro de 2016) e corrigido na versão 2.4 (20 de Abril de 2017).

- *cmd-switch-client.c:109: error:null dereference*, este erro foi introduzido na versão 1.9a (22 de Fevereiro de 2014), na qual foi reportado pelo CSA, e apenas foi identificado na versão 2.3 (29 de Setembro de 2016) e corrigido na versão 2.4 (20 de Abril de 2017).

4.2.3.2 Distribuição dos erros

Na Figura 4.7 estão representadas as percentagens de falsos positivos, verdadeiros positivos, falsos negativos e verdadeiros negativem identificados na última versão do Tmux.

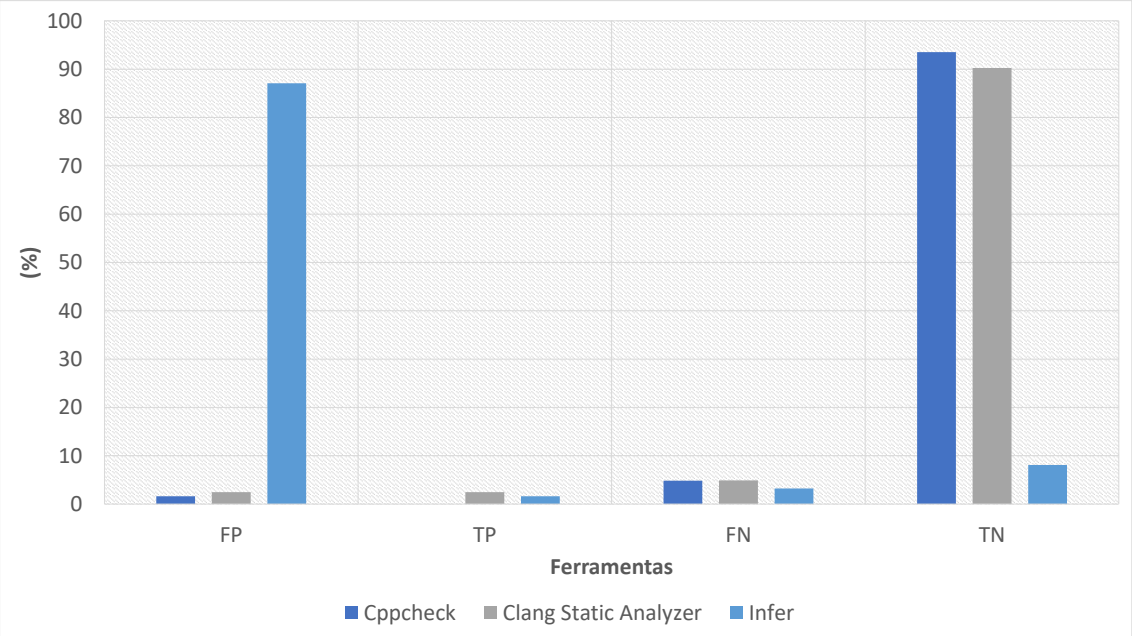


Figura 4.7: Distribuição dos vários tipos de erros identificados na versão 2.7 do Tmux pelas ferramentas.

O Cppcheck não identificou nenhum erro na última versão do Tmux e devolveu apenas 2 erros nas versões mais antigas do software. A ferramenta que devolve a maior quantidade de erros é o Infer, sendo que cerca de 89% são falsos positivos e cerca de 2% são verdadeiros positivos. A ferramenta que devolve a maior quantidade de verdadeiros positivos é o CSA, cerca de 3%, sendo que esta ferramenta devolve cerca de 3% de falsos positivos. Assim, para a última versão deste software a ferramenta que apresentou os melhores resultados foi o CSA, apesar de o Infer ter identificado uma maior quantidade de erros.

4.2.3.3 Eficácia e Precisão

Os valores de eficácia e precisão das 3 ferramentas no Tmux encontram-se representados no gráfico da Figura 4.8.

Neste projeto a ferramenta com o valor mais elevado de eficácia e precisão foi o CSA, sendo estes, respetivamente, cerca de 44% e 50%. O Infer obteve os valores de precisão mais reduzidos, uma vez que esta ferramenta devolveu uma grande quantidade de falsos positivos. O valor de precisão do Infer não ultrapassa os 3%.

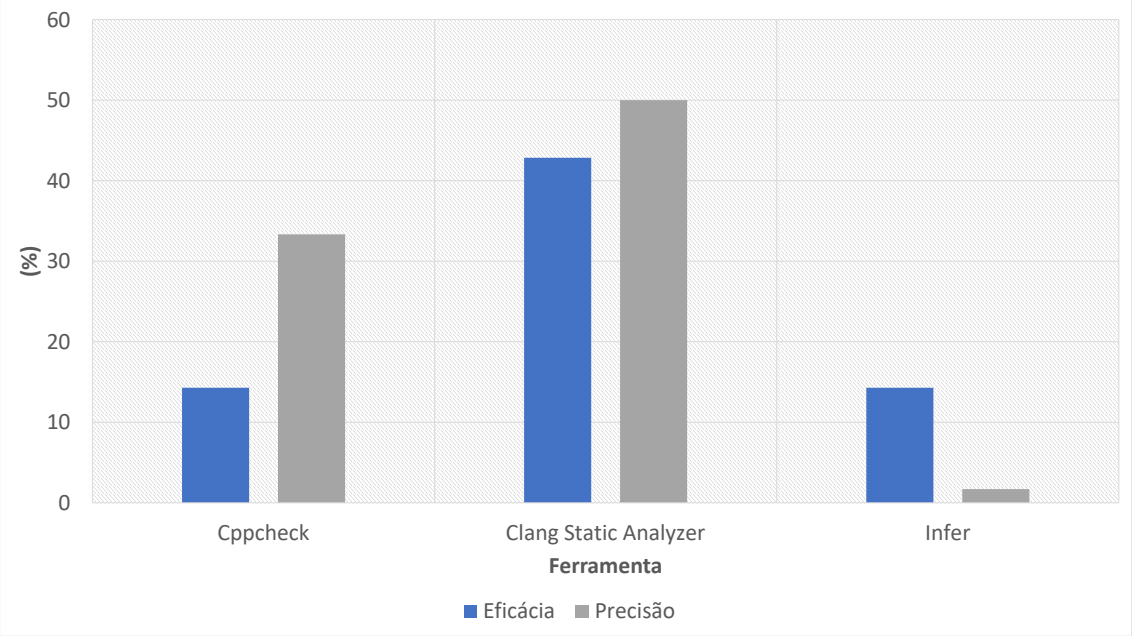


Figura 4.8: Valores de eficácia e precisão das 4 ferramentas no Tmux.

4.2.3.4 Tempo de execução

Na Figura 4.9 estão representados os tempos de execução das ferramentas no Tmux.

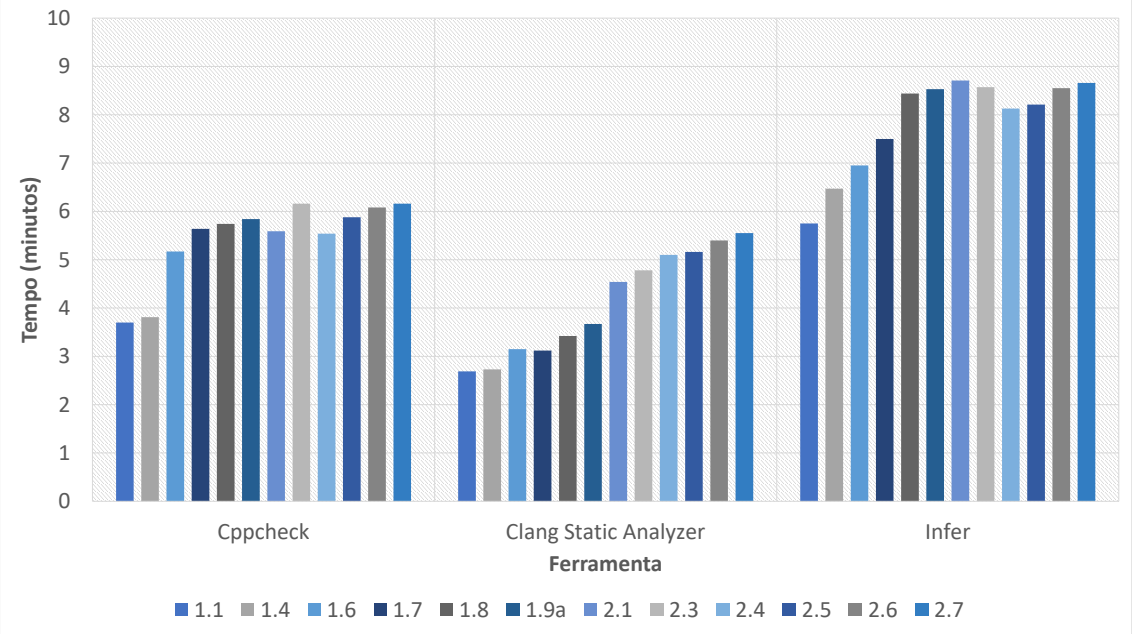


Figura 4.9: Tempo de execução das ferramentas na versão 2.7 do Tmux.

Contrariamente ao que se verificou com os projetos de menores dimensões (SDS e Beanstalkd), a execução do Cppcheck foi mais demorada do que a do CSA. No entanto, o CSA continua a devolver uma maior quantidade de resultados, nomeadamente de verdadeiros positivos. O Infer, tal como nos restantes softwares, é a ferramenta que demora

mais tempo a terminar a sua análise.

4.2.4 SQLite

Na versão 3.24.0 do SQLite foram identificados 205 erros pelas ferramentas. Nesta versão ainda não existem erros reportados, pelo que todos os erros identificados pelas ferramentas são novos. No entanto, apenas 21 erros são classificados como verdadeiros positivos. Os gráficos e tabelas desta secção não incluem a ferramenta Predator, pois esta não devolveu resultados conclusivos para nenhuma das versões do SQLite.

4.2.4.1 Tempo de permanência dos erros

Na Tabela 4.6 estão representados os tempos de permanência dos erros classificados como verdadeiros positivos e identificados por pelo menos uma ferramenta no SQLite. Nesta tabela estão apenas listados os erros reportados no versão 3.24.0 do SQLite, pelo que nenhum dos erros foi corrigido até ao momento.

4.2.4.2 Distribuição dos erros

As percentagens apresentadas na Figura 4.10 dizem respeito à última versão do SQLite.

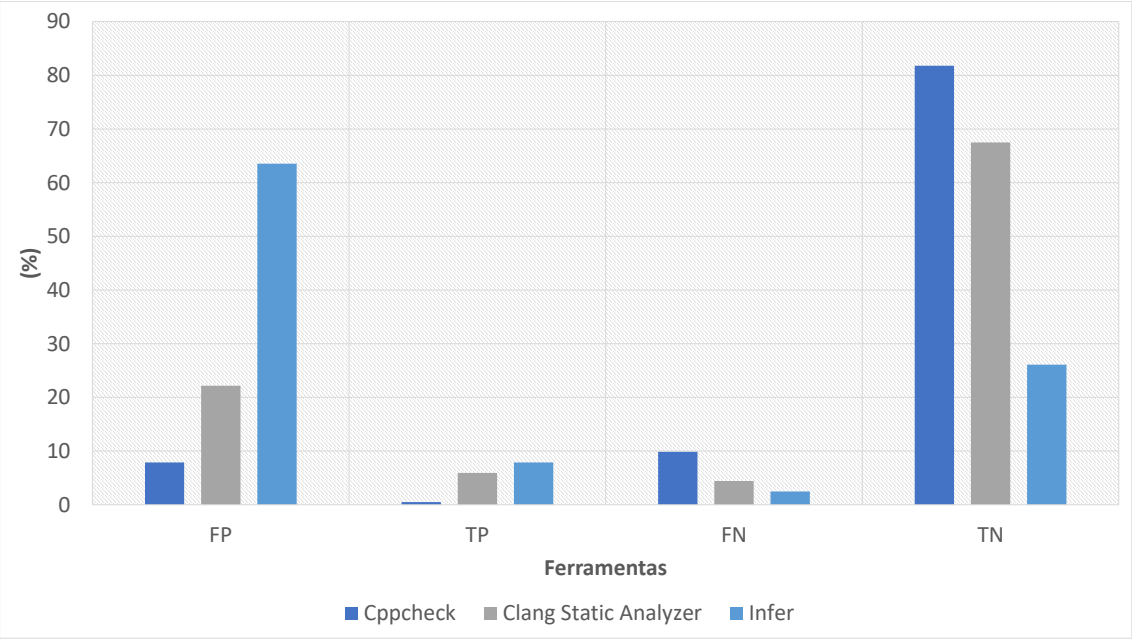


Figura 4.10: Distribuição dos vários tipos de erros identificados na versão 3.24.0 do SQLite pelas ferramentas.

O Predator não obteve resultados conclusivos para este projeto, pelo que não se encontra representado no gráfico da figura. A ferramenta que apresenta a maior percentagem de verdadeiros positivos é o Infer, sendo que esta ferramenta devolve também a maior quantidade de falsos positivos.

Tabela 4.6: Datas e tempo em meses decorrido desde a introdução até à correção dos erros no SQLite.

Erro		**Introduzido**	**Corrigido**	**Intervalo de tempo**
Localização	Tipo			(meses)
shell.c:13799	DS	24/out/17	x	x
shell.c:14953	DS	24/out/17	x	x
shell.c:3329	DS	04/jun/18	x	x
shell.c:4877	DS	04/jun/18	x	x
shell.c:585	ML	22/jan/18	x	x
sqlite3.c:104629	ND	22/jan/18	x	x
sqlite3.c:106296	ND	22/jan/18	x	x
sqlite3.c:112108	ND	04/jun/18	x	x
sqlite3.c:112344	ND	22/jan/18	x	x
sqlite3.c:113253	ND	22/jan/18	x	x
sqlite3.c:113375	ND	22/jan/18	x	x
sqlite3.c:120122	DS	22/jan/18	x	x
sqlite3.c:128851	ND	04/jun/18	x	x
sqlite3.c:130155	ND	04/jun/18	x	x
sqlite3.c:132810	ND	22/jan/18	x	x
sqlite3.c:135444	DS	04/jun/18	x	x
sqlite3.c:169136	ND	22/jan/18	x	x
sqlite3.c:199054	ND	04/jun/18	x	x
sqlite3.c:67330	DS	22/jan/18	x	x
sqlite3.c:75007	ND	22/jan/18	x	x
sqlite3.c:91920	ND	22/jan/18	x	x

O Cppcheck devolve a menor quantidade de falsos positivos, no entanto é também a ferramenta com a menor percentagem de verdadeiros positivos, tal como acontece nos restantes softwares analisados.

4.2.4.3 Eficácia e Precisão

Os valores de eficácia e precisão das 3 ferramentas no SQLite encontram-se representados no gráfico da Figura 4.11. No SQLite o valor da eficácia do Infer ultrapassa o do CSA, como se verificou na análise dos projetos de menor dimensão. Sendo que ambas as ferramentas

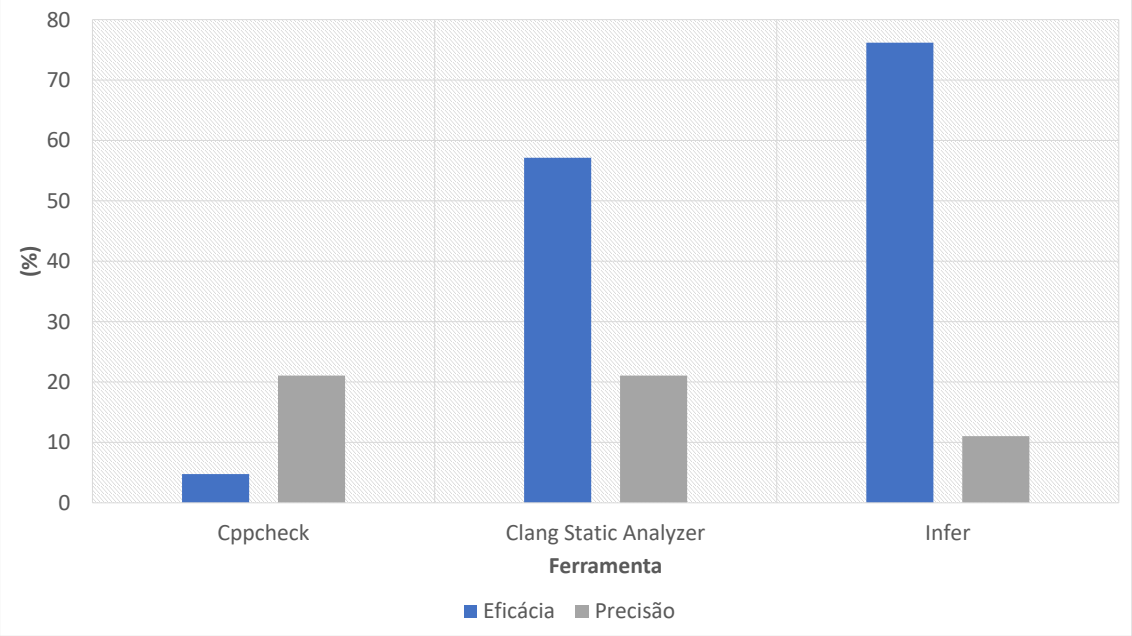


Figura 4.11: Valores de eficácia e precisão das 4 ferramentas no SQLite.

obtiveram valores de eficácia superiores a 50%. Por outro lado, o valor da precisão do Infer é inferior ao do CSA.

O Cppcheck, tal como se observou para os restantes projetos, obteve um valor de precisão mais elevado que o da eficácia. No entanto, para este projeto o valor da precisão do Cppcheck é considerada baixa, cerca de 20%.

4.2.4.4 Tempo de execução

Na Figura 4.12 estão representados os tempos de execução das ferramentas no SQLite.

O SQLite é um software de dimensão muito grande, pelo que, naturalmente, os valores do tempo de execução aumentaram relativamente aos restantes softwares. A análise do SQLite demorou entre 17 e 54 minutos.

Ao contrário do que se verificou para os restantes softwares, o CSA obteve os tempos de execução mais elevados, quanto que o Cppcheck obteve os valores mais reduzidos. Além disso, o Cppcheck obteve os valores de desvio padrão mais reduzidos, ou seja, o tempo de execução varia pouco ao longo das várias versões analisadas. No Infer e CSA observa-se um aumento significativo do tempo na análise da última versão.

4.3 Síntese

Neste capítulo foram apresentados e analisados os relatórios de resultados das ferramentas selecionadas. A partir da análise realizada foi possível perceber que a ferramenta que devolve a maior quantidade de erros é o Infer, seguido do CSA. O Predator, devolve uma grande quantidade de erros, no entanto, são na sua maioria falsos positivos, causados pelo

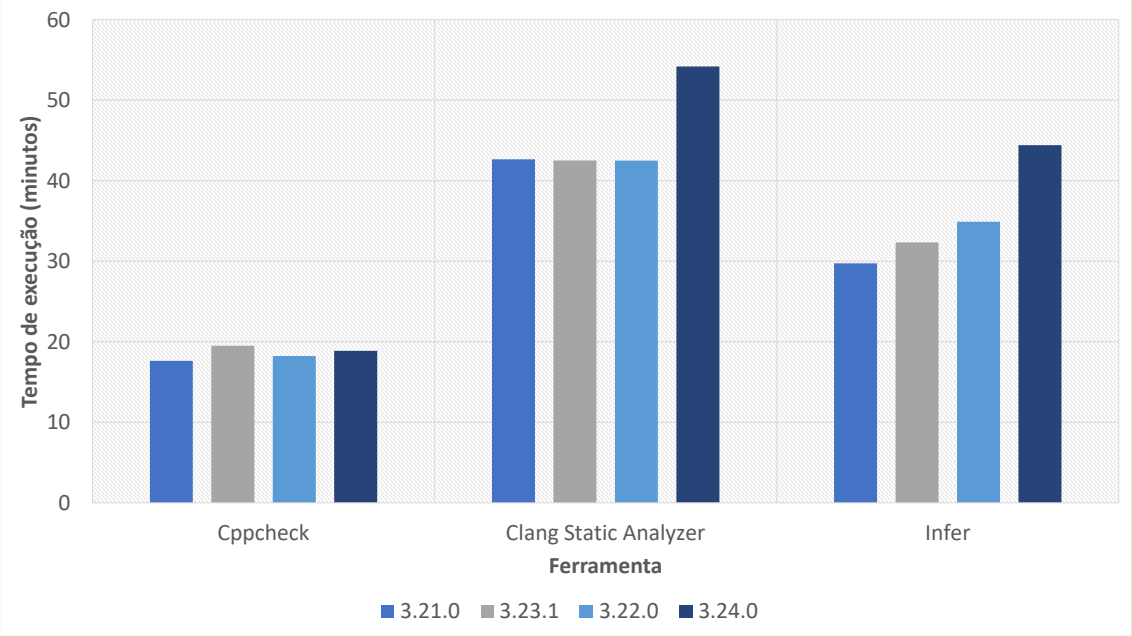


Figura 4.12: Tempo de execução das ferramentas na versão 3.24.0 do SQLite.

facto da ferramenta não estar preparada para analisar softwares complexos. O Cppcheck devolve a menor quantidade de erros, uma vez que esta ferramenta faz filtragem de falsos positivos e, portanto, ignora alguns dos erros reais.

Relativamente aos tempos de execução, o Cppcheck destaca-se pelo facto da sua análise ser a que demora menos tempo a ser concluída. Além disso, analisando o desvio padrão entre as várias versões dos softwares percebemos que o Cppcheck é a ferramenta com a menor variação de tempo de execução. Por outro lado, a análise realizada pelo Infer é a que, em média, demora mais tempo, seguida do CSA. Pelos motivos já referidos, não foi possível tirar conclusões relativamente aos tempos de execução do Predator. Assim, parece existir uma relação entre a quantidade de erros reportada e o tempo de execução da análise. A ferramenta que devolveu a maior quantidade de erros (falsos positivos e verdadeiros positivos) foi também a que demorou mais tempo a concluir a sua análise.

Capítulo 5

Uma comparação das ferramentas

No Capítulo 4 foram apresentados e analisados os resultados obtidos da execução das ferramentas nos projetos selecionados.

A ferramenta que devolveu a maior quantidade de erros foi o Infer, no entanto, a percentagem de falsos positivos ultrapassa a percentagem de verdadeiros positivos nos softwares de maior dimensão. Esta ferramenta é também a que apresenta os valores mais elevados de tempo de execução. O Predator obteve resultados pouco conclusivos em todos os projetos, uma vez que esta ferramenta não está preparada para softwares de elevada complexidade. O Cppcheck devolve a menor quantidade de erros, sendo que apresenta, normalmente, uma eficácia bastante baixa e precisão elevada. Os tempos de execução desta ferramenta são em média mais baixos que os das restantes ferramentas. Por fim, no CSA a percentagem de falsos positivos só é maior que a de verdadeiros positivos no SQLite, o software de maior dimensão. Além disso, o tempo de execução da análise desta ferramenta é em média menor que o do Infer.

No presente capítulo é feita uma comparação das ferramentas de análise estática relativamente à sua usabilidade, funcionalidade e limitações. A comparação foi elaborada com base nos dados obtidos da análise de resultados e é composta por várias componentes:

- Usabilidade;
- Funcionalidade;
- Limitações;

As ferramentas serão avaliadas, nas secções seguintes, para cada uma das componentes listadas.

5.1 Usabilidade

O Cppcheck dispõe de uma interface, no entanto, neste estudo foi utilizada a versão da linha de comandos. Esta ferramenta analisa todos os ficheiros de uma determinada diretoria e no final da sua análise devolve a lista de erros identificados. Os relatórios de resultados devolvidos pelo Cppcheck fornecem informação sobre a localização e descrição do erro reportado. No entanto, o tipo de erro nem sempre é explicito na descrição fornecida pela ferramenta. Na Listagem 5.1 está representado o relatório de um dos erros identificados pelo Cppcheck da análise da versão 1.0.0 do SDS.

Listagem 5.1: Exemplo dos relatórios de resultados devolvidos pelo Cppcheck.

```
1    [/home/fct/Software/first_v/sds-1.0.0/sds.c:159]: (error) Common realloc
2    mistake: 'sh' nulled but not freed upon failure
```

A análise do CSA foi realizada na linha de comandos através do *scan-build*. Este comando permite executar a ferramenta sobre o código fonte de um projeto como parte da execução de uma compilação regular. De facto, o *scan-build* sobrescreve as variáveis de ambiente CC e CXX para alterar a compilação e usar um compilador alternativo em vez daquele que normalmente iria construir o projeto. Este compilador falso executa o clang ou o gcc para compilar o código fonte e, em seguida, executa a ferramenta para analisar o código [11]. Após a conclusão da compilação, os resultados são apresentados numa página web onde podemos consultar o sumário de erros, organizado por tipo de erro, e a lista completa dos erros identificados. Além disso, para cada erro listado é possível consultar o seu relatório, que consiste no código fonte anotado com todos os passos que dão origem ao erro em questão. Desta forma, para além de ser referido o tipo de erro identificado é também possível compreender os passos seguidos durante a análise da ferramenta, facilitando a revisão manual do código. Na Figura 5.1 (página 63) está representado um relatório de resultados devolvidos pelo CSA da análise da versão 1.0.0 do SDS.

O Infer foi executado a partir da linha de comandos e, tal como o CSA, realiza a sua análise como parte da compilação regular do código. Os relatórios devolvidos pelo Infer fornecem informação sobre o número de erros encontrados, lista de erros, que fornece informação sobre o tipo, descrição e a parte do código onde o erro foi reportado, e sumário dos relatórios, com o número de erros de cada tipo identificados. Na Listagem 5.2 (página 54) está representado um relatório de resultados devolvido pelo Infer da análise da versão 1.0.0 do SDS.

Listagem 5.2: Exemplo dos relatórios de resultados devolvidos pelo Infer.

```
1    Found 2 issues
2
3  sds.c:160: error: NULL_DEREFERENCE
4    pointer `sh` last assigned on line 159 could be null and is dereferenced at
5    line 160, column 5.
6    158.       sh = (void*) (s-sizeof *sh);;
7    159.       sh = realloc(sh, sizeof *sh+sh->len+1);
```

```
  160. >     sh->free = 0;
  161.       return sh->buf;
  162.   }

sds.c:817: error: NULL_DEREFERENCE
  pointer `x` last assigned on line 816 could be null and is dereferenced by
  call to `memcmp()` at line 817, column 9.
  815.           sdsfree(x);
  816.           x = sdscatprintf(sdsempty(),"\texttt{\%}d",123);
  817. >         test_cond("sdscatprintf() seems working in the base case",
  818.               sdslen(x) == 3 && memcmp(x,"123\0",4) ==0)
  819.

Summary of the reports

  NULL_DEREFERENCE: 2
```

Por fim, o Predator foi utilizado na sua versão de *plug-in* do gcc. Para que a aplicação deste *plug-in*, numa construção de um projeto arbitrário, seja totalmente transparente é necessário definir certas variáveis de ambiente, processo feito de forma automática utilizando o *script register-paths.sh*. Após concluído este processo basta utilizar a opção *-fplugin=libsl.so* do gcc e compilar normalmente o projeto. Os relatórios de resultados devolvidos pelo Predator fornecem informação sobre a localização, tipo e descrição do erro. Além disso, esta ferramenta permite imprimir o estado da memória do programa à medida que é feita a análise. Para isso basta ativar a impressão do SMG através dos comandos da Listagem 5.3.

Listagem 5.3: Comandos para ativar a impressão de SMGs na ferramenta Predator.

```
export  SL_PLOT=1
export  SVG_VIEWER=chromium-browser  # Adjust for your SVG viewer
```

Na Listagem 5.4 está representado uma parte do relatório de resultados devolvido pelo Predator da análise da versão 1.0.0 do SDS.

Listagem 5.4: Exemplo dos relatórios de resultados devolvidos pelo Predator.

```
Trying to compile sds.c ... OK
Running Predator ...
sl/cl_symexec.cc:156: warning: main() not found at global scope
sds.c:779:25: error: invalid dereference
sds.c:782:5: warning: memory leak detected while destroying a variable on stack
sds.c:107:25: error: invalid dereference
sds.c:76:41: error: invalid dereference
sds.c:77:5: warning: memory leak detected while destroying a variable on stack
sds.c:203:5: error: invalid dereference
sds.c:217:31: error: invalid dereference
sds.c:632:11: error: invalid dereference
sds.h:49:14: error: invalid dereference
sds.c:747:18: note: from call of sdslen()
```

```
14 sds.c:746:5: note: from call of sdsmapchars()
```

5.2 Funcionalidade

Nas tabelas 5.1, 5.2, 5.3 e 5.4 estão representados a quantidade de cada tipo de erro identificada pelas ferramentas, a quantidade erros reportados no repositório que aquelas não conseguiram identificar e o total de erros de cada tipo identificados nos vários projetos analisados. Além disso, estas tabelas fornecem-nos informação sobre a quantidade de erros identificados por apenas uma ferramenta, desta forma é possível determinar se as ferramentas obtêm resultados complementares.

Tabela 5.1: Síntese de erros identificados no SDS.

Tipo de erro	**Cppcheck**		**CSA**		**Infer**		**Predator**		**Não identificados p/ ferramentas**	**Identificados apenas p/ uma ferramenta**	**Total**
	FP	TP	FP	TP	FP	TP	FP	TP			
ML	0	1	0	0	0	0	3	0	4	1(Cppcheck)	4
InvD	0	0	0	0	0	0	8	0	0	0	8
ND	0	0	0	2	0	4	1	0	1	4(Infer) + 2(CSA)	7
InvF	0	0	2	0	0	0	1	0	0	0	3
UAF	0	0	0	0	0	0	0	0	0	0	0
UV	0	0	0	0	0	0	0	0	0	0	0
DS	0	0	0	1	1	1	0	0	0	0	3
OB	0	0	0	0	0	0	0	0	0	0	0
SegF	0	0	0	0	0	0	0	0	0	0	0
RL	0	0	0	0	0	0	0	0	0	0	0
DP	0	0	0	0	0	0	0	0	0	0	0
BO	0	0	0	0	0	0	0	0	0	0	0

O Cppcheck provou-se útil na identificação de erros do tipo fuga de memória. Esta ferramenta foi responsável por reportar uma fuga de memória do SDS, que não foi identificada por nenhuma outra ferramenta, uma fuga de memória no Beanstalkd, que foi também identificada pelo Predator, e uma fuga de memória no SQLite, que foi simultaneamente identificada pelo Infer e CSA. Esta ferramenta foi também responsável por identificar um erro do tipo fuga de recursos no Beanstalkd, que não foi identificado por nenhuma outra ferramenta.

O CSA foi responsável pela identificação de 10 erros do tipo desreferência nula, sendo que todos eles foram reportados exclusivamente por esta ferramenta. O CSA também se provou útil na identificação de *dead stores*, por outro lado, esta ferramenta reporta apenas

Tabela 5.2: Síntese de erros identificados no Beanstalkd.

Tipo de erro	Cppcheck		CSA		Infer		Predator		Não identificados p/ ferramentas	Identificados apenas p/ uma ferramenta	Total
	FP	TP	FP	TP	FP	TP	FP	TP			
ML	0	1	0	0	4	1	2	2	7	1(Infer) + 1(Predator)	10
InvD	0	0	0	0	0	0	126	0	0	0	126
ND	0	0	0	3	0	11	1	0	0	10(Infer) + 3(CSA)	15
InvF	0	0	0	0	0	0	1	0	3	0	1
UAF	0	0	0	0	0	0	0	0	0	0	0
UV	0	0	0	0	0	0	0	0	0	0	0
DS	0	0	0	3	6	4	0	0	0	1(Infer)	13
OB	0	0	0	0	0	0	0	0	0	0	0
SegF	0	0	0	0	0	0	0	0	2	0	0
RL	0	1	0	0	2	1	0	0	0	1(Infer) + 1(Cppcheck)	4
DP	0	0	0	0	0	0	0	0	1	0	0
BO	0	0	0	0	0	0	0	0	1	0	0

um erro do tipo fuga de memória, sendo que este foi identificado simultaneamente pelo Cppcheck e Infer, e nenhum erro do tipo fuga de recursos.

O Infer é a ferramenta que devolve a maior quantidade de erros. Apesar de devolver uma elevada percentagem de falsos positivos, esta ferramenta foi a única capaz de identificar 22 erros do tipo desreferência nula, 2 do tipo fuga de memória, 1 do tipo fuga de recursos e 2 do tipo *dead store* nos vários softwares analisados.

O Predator é uma ferramenta útil para projetos pequenos e pouco complexos, como ficou óbvio após algumas tentativas de análise dos softwares selecionados. A ferramenta é capaz de identificar os vários tipos de erros de memória nos quais estamos interessados, no entanto, nos softwares estudados não foi capaz de devolver resultados conclusivos na maioria dos casos. Ainda assim, o Predator foi responsável pela identificação exclusiva de 1 erro do tipo fuga de memória no Beanstalkd.

A partir desta experiência foi possível identificar padrões de erros e construir exemplos mínimos capazes de reproduzir os resultados observados nos programas analisados. Na Tabela 5.5 (página 62) estão representados os padrões identificados para cada tipo de erro. Os exemplos mínimos e respetivos relatórios de resultados podem ser consultados no Apêndice B.

A ferramenta Infer é a única que identifica a possibilidade da ocorrência de uma desreferência nula quando não é feita a verificação dos resultados das operações de alocação de memória. Se as funções `malloc`, `calloc` e `realloc` falharem, estas retornam o valor `NULL` e, portanto, a desreferência dessa variável poderá gerar um erro. O Infer é também a única ferramenta que identifica que o valor de um endereço não está a ser utilizado (i.e., *dead store*), caso esse valor seja 0 ou *NULL*. A ferramenta CSA não considera esta situação

Tabela 5.3: Síntese de erros identificados na versão 2.7 do Tmux.

Tipo de erro	Cppcheck		CSA		Infer		Não identificados p/ ferramentas	Identificados apenas p/ uma ferramenta	Total
	FP	TP	FP	TP	FP	TP			
ML	0	0	0	0	10	1	0	1(Infer)	11
InvD	0	0	0	0	0	0	0	0	0
ND	0	0	1	1	33	0	1	1(CSA)	35
InvF	0	0	0	0	0	0	0	0	0
UAF	0	1	2	2	4	0	0	2(CSA) + 1(Cppcheck)	9
UV	1	0	0	0	1	0	0	0	2
DS	0	0	0	0	9	0	0	0	9
OB	0	0	0	0	0	0	1	0	0
SegF	0	0	0	0	0	0	0	0	0
RL	0	0	0	0	1	0	0	0	1
DP	0	0	0	0	0	0	0	0	0
BO	0	0	0	0	0	0	0	0	0

como um erro, uma vez que considera que o valor 0 ou *NULL* ao ser atribuído à variável na sua inicialização não é um valor não utilizado. O Infer (até à versão 0.13.1) não conseguia identificar qualquer tipo de erro de memória relacionado com a utilização de uma variável alocada usando o padrão *sizeof(*ptr)*, e.g., `ptr = malloc(sizeof(*ptr))`, mas na sua versão mais recente (versão 0.14.0, lançada no dia 1 de Maio de 2018) este defeito já foi corrigido. No Apêndice B encontra-se um exemplo mínimo da situação descrita. O Infer reporta um falso positivo do tipo variável não inicializada quando a variável em questão está a ser inicializada dentro de um ciclo infinito ou condição que é sempre verdadeira. Ou seja, a ferramenta não percebe que a variável vai ser sempre inicializada. Este problema foi reportado no repositório do Infer e pode ser consultado em https://github.com/facebook/infer/issues/974.

Durante a análise do SQLite verificou-se que este software utiliza asserções para realizar verificações no código. No entanto, o CSA e o Infer não reconhecem esse mecanismo e reportam vários erros, que foram classificados como falsos positivos. Isto é, se for utilizada uma asserção para verificar se o valor de um apontador é diferente de `NULL`, antes de este ser desreferenciado, ambas as ferramentas vão reportar um erro do tipo desreferência nula. Na Secção B.3 do Apêndice B está representado um exemplo da situação descrita.

Apesar de o Infer ser a única ferramenta que sistematicamente reporta a não verificação das chamadas a funções de alocação de memória, observou-se que também o Cppcheck, em algumas situações, reporta este tipo erros. No entanto, os erros reportados

Tabela 5.4: Síntese de erros identificados na versão 3.24.0 do SQLite.

Tipo de erro	Cppcheck		CSA		Infer		Não identificados p/ ferramentas	Identificados apenas p/ uma ferramenta	Total
	FP	TP	FP	TP	FP	TP			
ML	0	1	0	1	8	1	0	0	11
InvD	0	0	0	0	0	0	0	0	
NuD	12	0	37	4	23	8	0	8(Infer) + 4(CSA)	84
InvF	0	0	0	0	0	0	0	0	0
UAF	0	0	0	0	0	0	0	0	0
UV	2	0	0	0	61	0	0	0	63
DS	0	0	8	7	36	7	0	1(Infer) + 1(CSA)	58
OB	0	0	0	0	0	0	0	0	0
SegF	0	0	0	0	0	0	0	0	0
RL	2	0	0	0	1	0	0	0	3
DP	0	0	0	0	0	0	0	0	0
BO	0	0	0	0	0	0	0	0	0

pelo Cppcheck e Infer são diferentes. O Cppcheck identifica uma possível fuga de memória, enquanto que o Infer identifica uma desreferência nula. Isto acontece quando é feita uma atribuição sem que seja verificado o resultado da chamada à função `realloc`, ficando assim a memória, anteriormente atribuída, inacessível no caso de esta operação falhar. No entanto, o Cppcheck revelou-se pouco eficaz na identificação de erros quando existem chamadas a funções. Quando o apontador é inicializado com o valor `NULL` e a desreferência é realizada dentro da mesma função, o erro é reportado pelas 4 ferramentas. Por outro lado, quando a desreferência é feita noutra função o Cppcheck não consegue identificar que existe um erro. Além disso, esta ferramenta não é capaz de identificar erros do tipo fuga de memória quando as operações de manipulação não são feitas de forma explicita e quando o erro ocorre numa estrutura com apontadores. Isto é, a fuga de memória ocorre, porque a memória atribuída à estrutura é libertada antes da memória atribuída ao apontador e, portanto, essa memória fica inacessível.

5.3 Limitações

O Cppcheck faz filtragem de erros, portanto, devolve um baixa percentagem de falsos positivos, verdadeiros positivos, e, consequentemente, uma elevada percentagem de falsos negativos. De facto, esta ferramenta reporta uma quantidade muito reduzida de erros, em qualquer dos projetos analisados. Além disso, esta ferramenta não reconhece muitos

dos padrões de erros apresentados na Tabela 5.5. Relativamente aos relatórios devolvidos pelo Cppcheck estes não contêm sumário (i.e., quantidade e tipo de erros) e fornecem pouca informação sobre o tipo de erros identificados.

O CSA é a ferramenta que identifica o menor número de fugas de memória e fugas de recursos. Para além disso, tal como o Cppcheck, esta ferramenta não reconhece alguns dos padrões de erros estudados.

O Infer é a ferramenta que, na maioria dos softwares analisados, demora mais tempo a realizar a sua análise, sendo que é também a ferramenta que devolve a maior quantidade de erros. No entanto, é a ferramenta que devolve a maior percentagem de falsos positivos.

Para concluir, as limitações do Predator são as seguintes: i) A análise não está pronta para projetos complexos; ii) O programa analisado deve ser fechado; iii) A verificação modular não é suportada; iv) Não suporta programas recursivos; v) A ferramenta funciona apenas no Linux (Ubuntu 14.04).

Durante a análise da versão 2.4 do Tmux verificou-se que o tempo de execução do Predator estava a atingir valores demasiado elevados, pelo que a sua analise foi interrompida. Observando os relatórios de resultados, obtidos até ao momento da interrupção, verificou-se que a ferramenta tinha entrado num ciclo infinito. Na Listagem 5.5 está representado uma parte do relatório produzido pelo Predator. Nesta listagem podemos ver que a ferramenta ignora sistematicamente várias chamadas a funções, que considera externas, entrando assim num ciclo.

Listagem 5.5: Parte do relatório de resultados devolvido pelo Predator na análise da versão 2.4 do Tmux.

```
1     /home/ps.monteiro/predator/sl/cl_symexec.cc:156: warning: main() not found
2     at global scope [internal location] [-fplugin=libsl.so]
3  key-bindings.c:27:1: error: invalid dereference [-fplugin=libsl.so]
4  key-bindings.c:410:2: error: invalid dereference [-fplugin=libsl.so]
5  key-bindings.c:28:1: error: invalid dereference [-fplugin=libsl.so]
6  key-bindings.c:27:1: error: invalid dereference [-fplugin=libsl.so]
7  key-bindings.c:28:1: error: invalid dereference [-fplugin=libsl.so]
8  key-bindings.c:386:11: warning: ignoring call of undefined function:
9  cmd_string_parse() [-fplugin=libsl.so]
10 key-bindings.c:389:3: warning: ignoring call of undefined function:
11 cmdq_get_command() [-fplugin=libsl.so]
12 key-bindings.c:389:3: warning: ignoring call of undefined function:
13 cmdq_append() [-fplugin=libsl.so]
14 key-bindings.c:390:3: warning: ignoring call of undefined function:
15 cmd_list_free() [-fplugin=libsl.so]
16 key-bindings.c:386:11: warning: ignoring call of undefined function:
17 cmd_string_parse() [-fplugin=libsl.so]
18 key-bindings.c:389:3: warning: ignoring call of undefined function:
19 cmdq_get_command() [-fplugin=libsl.so]
20 key-bindings.c:389:3: warning: ignoring call of undefined function:
21 cmdq_append() [-fplugin=libsl.so]
22 key-bindings.c:390:3: warning: ignoring call of undefined function:
```

```
cmd_list_free() [-fplugin=libsl.so]
key-bindings.c:386:11: warning: ignoring call of undefined function:
cmd_string_parse() [-fplugin=libsl.so]
key-bindings.c:389:3: warning: ignoring call of undefined function:
cmdq_get_command() [-fplugin=libsl.so]
key-bindings.c:389:3: warning: ignoring call of undefined function:
cmdq_append() [-fplugin=libsl.so]
key-bindings.c:390:3: warning: ignoring call of undefined function:
cmd_list_free() [-fplugin=libsl.so]
key-bindings.c:386:11: warning: ignoring call of undefined function:
cmd_string_parse() [-fplugin=libsl.so]
key-bindings.c:389:3: warning: ignoring call of undefined function:
cmdq_get_command() [-fplugin=libsl.so]
key-bindings.c:389:3: warning: ignoring call of undefined function:
cmdq_append() [-fplugin=libsl.so]
key-bindings.c:390:3: warning: ignoring call of undefined function:
cmd_list_free() [-fplugin=libsl.so]
```

Tabela 5.5: Padrões de erros.

Padrão	**Erro**	**Ferramenta**			
		Cppcheck	CSA	Infer	Predator
Apontador inicializado com o valor `NULL` e desreferenciado na mesma função.	ND	✓	✓	✓	✓
Apontador inicializado com o valor `NULL` noutra função e desreferenciado.	ND	x	✓	✓	✓
Falta de verificação após uma alocação de memória.	ND	x	x	✓	x
Utilização de asserções	ND	x	✓	✓	x
O valor escrito no endereço é inicializado com um valor diferente de `NULL` e 0 e nunca é usado.	DS	x	✓	✓	x
O valor escrito no endereço é inicializado com o valor `NULL` ou 0 e nunca é usado.	DS	x	x	✓	x
O valor do apontador é alterado através da função `realloc`, sem garantir que esta tem sucesso (sem chamadas a funções).	ML	✓	x	✓	x
O valor do apontador é alterado através da função `realloc`, sem garantir que esta tem sucesso (com chamadas a funções).	ML	x	x	✓	x
Alocação e libertação de memória explicita.	ML	✓	✓	✓	✓
Alocação e libertação de memória não explicita.	ML	x	✓	✓	✓
Libertação da estrutura antes da libertação da memória alocada para apontadores que fazem parte da mesma.	ML	x	✓	✓	✓
A variável é inicializada dentro de um ciclo infinito, cuja condição é sempre verdadeira (i.e., `for(;;)` ou `while(true)` ou `while(1)`).	UV	x	x	✓	x
A variável é inicializada dentro de uma condição que é sempre verdadeira (i.e., `if(true)` ou `if(1)`).	UV	x	x	✓	x
O valor passado como argumento à função de alocação de memória (`malloc` ou `calloc`) é do tipo `*ptr`.	ND ML InvF	x	✓	x	✓

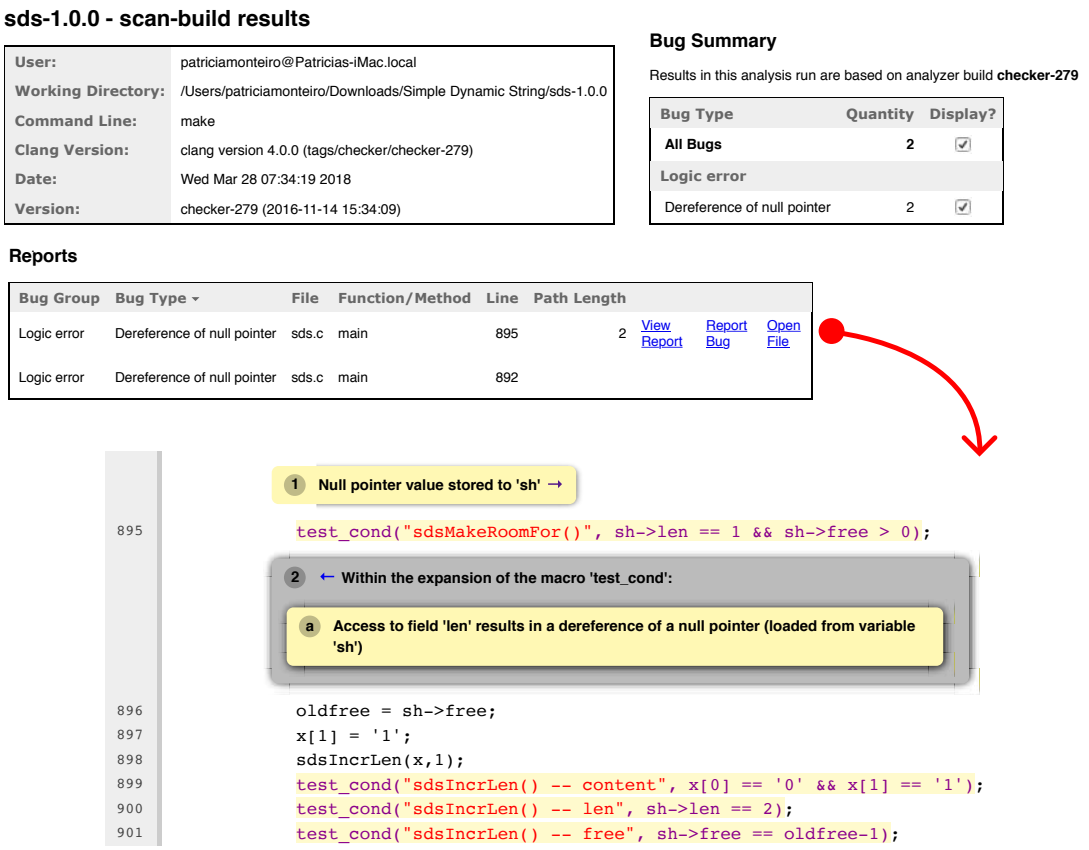


Figura 5.1: Exemplo dos relatórios de resultados devolvidos pelo CSA.

Capítulo 6

Conclusões

O objetivo desta dissertação é fazer uma análise comparativa de 4 ferramentas de análise estática: i) Cppcheck; ii) Clang Static Analyzer (CSA); iii) Infer; iv) Predator. As ferramentas foram aplicadas a um conjunto de projetos escritos na linguagem C/C++, com diferentes dimensões e complexidades: i) Simple Dynamic String (SDS); ii) Beanstalkd; iii) Tmux; iv) SQlite. O foco desta experimentação foi comparar a eficácia e precisão na identificação de erros de memória, bem como medir o tempo de execução e a percentagem de falsos positivos, verdadeiros positivos, falsos negativos e verdadeiros negativos reportados pelas ferramentas. No Capítulo 4 foram apresentados e analisados os relatórios de resultados obtidos da aplicação das ferramentas aos projetos selecionados. No Capítulo 5 foi feita uma comparação das ferramentas de análise estática, a nível da sua usabilidade, funcionalidade e limitações.

No total foram analisados 4 programas, 43 versões e 486 erros, dos quais 78 são verdadeiros positivos. O Infer foi responsável por identificar a maior percentagem dos verdadeiros positivos reportados, que corresponde a 50%, sendo que identificou 3 das 16 fugas de memória, 23 das 35 desreferências nulas, 12 dos 13 *dead stores* e 1 das 2 fugas de recursos. A ferramenta que devolveu a segunda maior percentagem de verdadeiros positivos foi o CSA, que identificou 28% destes erros. Esta ferramenta reportou 1 das 16 fugas de memória, 10 das 35 desreferências nulas, 2 das 3 utilizações após libertação de memória e 11 dos 13 *dead stores*. O Cppcheck identificou 5% dos verdadeiros positivos, dos quais reportou 3 das 16 fugas de memória, 1 das utilizações após libertação de memória e 1 das 1 fugas de recursos. Por fim, o Predator identificou 3% dos verdadeiros positivos, que corresponde à identificação de 2 das 16 fugas de memória. No gráfico da Figura 6.1 estão representadas as percentagens de erros reais, de cada tipo, identificadas pelas ferramentas. Neste gráfico estão também representadas as percentagens de erros que fazem parte do históricos de erros dos repositórios dos softwares, mas não foram

identificados por nenhuma ferramenta.

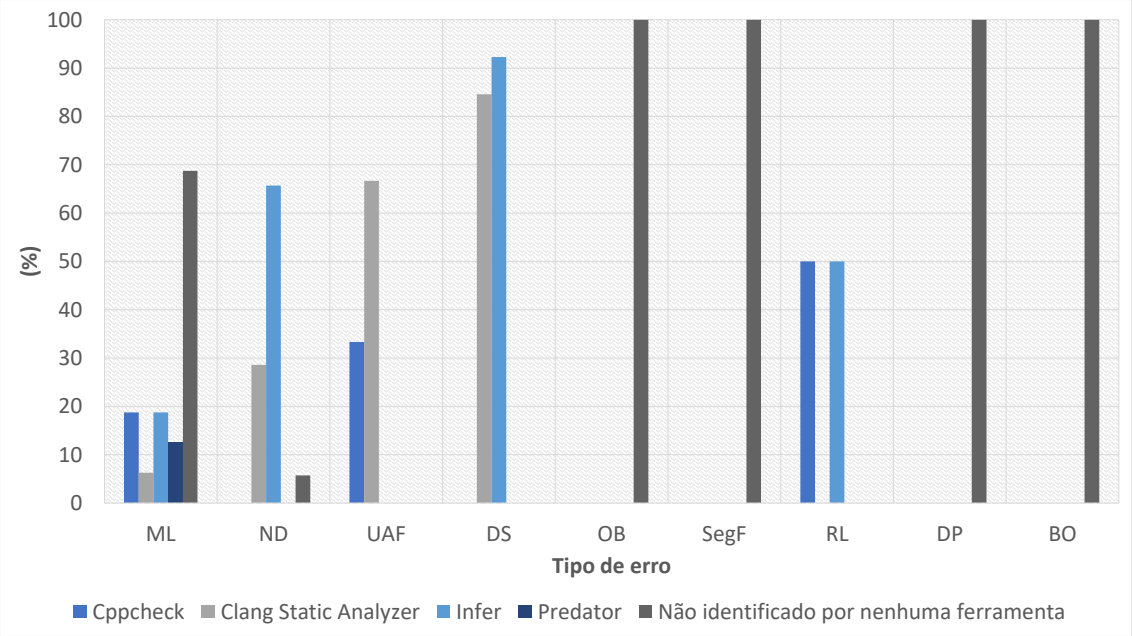


Figura 6.1: Percentagem dos vários tipos de erros identificados nos 4 softwares pelas ferramentas.

No diagrama da Figura 6.2 estão representadas as percentagens de erros reais de cada tipo identificadas por cada ferramenta. Neste diagrama as interseções correspondem aos erros identificados por duas ou mais ferramentas, as zonas fora das interseções correspondem aos erros identificados exclusivamente por uma ferramenta especifica. Por fim, as percentagens representadas fora do diagrama correspondem aos erros que não foi reportados por nenhuma das ferramentas. Como se pode observar na figura, a percentagem de erros identificados por mais que uma ferramenta é muito reduzido, sendo que nenhum erro foi identificado por todas as ferramentas. Como já tinha sido confirmado pelos dados apresentados nas tabelas 5.1, 5.2, 5.3 e 5.4 do Capítulo 4, cada ferramenta identificou, pelo menos, um erro que nenhuma das outras identificou. Portanto, os resultados deste estudo confirmam que as 4 ferramentas são úteis. Existe benefícios em usar várias ferramentas de análise estática em combinação, a fim de identificar uma maior quantidade de erros reais.

É também importante salientar que as utilização destas ferramentas não pesa significativamente no desenvolvimento de software, pois os tempos que cada uma leva a analisar os projetos escolhidos são muito reduzidos, vão de alguns segundos, no caso do SDS, a um máximo que não ultrapassada uma hora, no caso do SQLite. Os tempos de execução estão relacionados com a forma como as ferramentas constroem a sua representação do código, sendo que apesar de um ferramenta demorar mais tempo a realizar a sua análise não significa que a mesma seja mais ou menos eficiente.

Vários dos erros encontrados pelas ferramentas estiveram presentes nos softwares por longos períodos de tempo, como foi referido no Capítulo 4. A deteção destes erros pelas

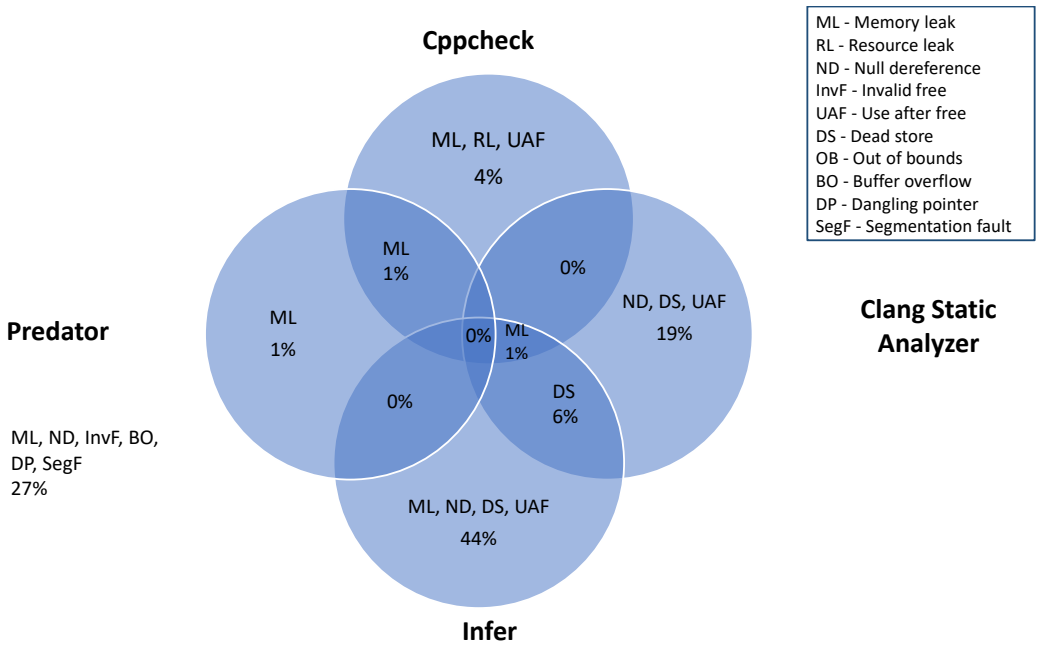


Figura 6.2: Diagrama de *Venn* das percentagens de erros reais identificados nos 4 projetos analisados.

ferramentas foi muito rápida e a sua validação através de revisão manual levou poucas horas, exceto no caso do Predator. Esta ferramenta devolveu uma grande quantidade de falsos positivos, pois não está preparada para projetos muito complexos. Por este motivo, exige demasiado o esforço extra para verificar todos os seus resultados. Assim, consideramos que a utilização do Predator, nos projetos estudados, não é muito vantajosa.

Os exemplos mínimos construidos para simular os padrões de erros identificados pelas ferramentas, confirmam que muitos dos erros de memória são instância de um conjunto limitado de padrões. Ou seja, a origem do problema não varia muito. Além disso, por norma, a resolução destes erros não é muito complexa.

Concluindo, consideramos que a utilização de ferramentas de análise estática, durante o processo de desenvolvimento de software, é vantajosa para evitar erros de memória. Em súmula, esta experiência permitiu comprovar o impacto que a utilização de ferramentas de análise estática pode ter na verificação de software. De facto, este tipo de ferramentas está já a ser incluído no processo de desenvolvimento de alguns programas de grande dimensão e relevância, como é o caso do LibreOffice[1]. Atualmente, este software integra no seu processo de desenvolvimento duas ferramentas de análise estática, o Cppcheck [39], utilizado nesta experiência, e o Coverity [38], uma marca de produtos de desenvolvimento de software da empresa Synopsys[2], que oferece um serviço de análise estática gratuito chamado Coverity Scan. Segundo o relatório disponibilizado por este serviço , até ao momento foram analisadas mais de 6 milhões de linhas de código do projeto LibreOffice, onde foram identificados mais de 25 mil erros, dos quais foram corrigidos 99% [53]. No

[1] https://github.com/LibreOffice/core
[2] https://www.synopsys.com/

caso do Cppcheck, segundo o último relatório disponível, referente ao dia 27 de Janeiro de 2018, foram identificados através desta ferramenta cerca de 6 mil erros no repositório do LibreOffice [54]. O caso do LibreOffice é um bom exemplo dos benefícios da integração de ferramentas de análise estática no processo de desenvolvimento do software. Estas ferramentas, para além de permitirem a deteção de erros numa fase inicial do desenvolvimento, permitem provar a ausência de erros. Se uma ferramenta de análise estática foi construída para identificar um determinado padrão de erro, então vai ser capaz de provar que esse padrão nunca se verifica, ou caso contrário, irá devolver todas as situações em que este ocorre.

Bibliografia

[1] *"clang"C Language Family Frontend for LLVM*. URL: http://clang.llvm.org/.

[2] *AdLint::Advanced Lint*. URL: http://adlint.sourceforge.net/.

[3] F. E. Allen. "Control Flow Analysis". Em: *SIGPLAN Not.* 5.7 (1970). URL: %5Curl%7Bhttp://doi.acm.org/10.1145/390013.808479%7D.

[4] *Automated code reviews & code analytics|Codacy*. URL: https://www.codacy.com/.

[5] R. Bagnara, M. Carlier, R. Gori e A. Gotlieb. "Symbolic Path-Oriented Test Data Generation for Floating-Point Programs". Em: *ICST*. IEEE, 2013.

[6] T. Ball. "The Concept of Dynamic Analysis". Em: *SIGSOFT Softw. Eng. Notes* 24.6 (1999). URL: %5Curl%7Bhttp://doi.acm.org/10.1145/318774.318944%7D.

[7] T. Ball, B. Cook, V. Levin e S. Rajamani. "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft". Em: *iFM*. Vol. 2999. LNCS. 2004, pp. 1–20.

[8] C. Bolduc. "Lessons Learned: Using a Static Analysis Tool within a Continuous Integration System". Em: *ISSREW*. IEEE, 2016.

[9] E. C. Brewer. "The Single-Responsibility". Em: *Agile Principles, Patterns, And Practices In C* (2007).

[10] C. Calcagno e D. Distefano. "Infer: An Automatic Program Verifier for Memory Safety of C Programs". Em: *NFM*. Vol. 6617. LNCS. Springer, 2011.

[11] *Clang Static Analyzer*. URL: https://clang-analyzer.llvm.org/.

[12] *CodeSonar – Static Analysis SAST Software for Secure SDLC | GrammaTech*. URL: https://www.grammatech.com/products/codesonar.

[13] L. Correnson et al. *Frama-C User Manual*. https://frama-c.com/download.html. CEA LIST, 2017.

[14] P. Cousot e R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". Em: *POPL*. ACM, 1977.

[15] *Coverity Scan - Static Analysis*. https://scan.coverity.com/projects/beanstalkd. 2013.

[16] *Cppdepend::C/C++ Static Analysis and Code Quality tool*. URL: https://www.cppdepend.com/.

[17] *Cpplint*. URL: https://github.com/google/styleguide/tree/gh-pages/cpplint.

[18] D. Delmas e J. Souyris. "Astrée: From Research to Industry". Em: *SAS*. Vol. 4634. LNCS. Springer, 2007.

[19] K. Dudka, P. Peringer e T. Vojnar. "Byte-Precise Verification of Low-Level List Manipulation". Em: *SAS*. Vol. 7935. LNCS. Springer, 2013.

[20] D. A. Duffy. *Principles of Automated Theorem Proving*. John Wiley & Sons, 1991.

[21] *Dynamic Analysis vs. Static Analysis*. https://software.intel.com/en-us/inspector-user-guide-linux-dynamic-analysis-vs-static-analysis.

[22] D. Evans e D. Larochelle. "Improving security using extensible lightweight static analysis". Em: *IEEE Software* 19.1 (2002).

[23] D. Evans, J. Guttag, J. Horning e Y. M. Tan. "LCLint: A Tool for Using Specifications to Check Code". Em: *FSE*. ACM, 1994.

[24] E. Farchi. *Review Moderator Workshop*. 2015.

[25] A. Fehnker et al. "Goanna—A Static Model Checker". Em: *FMICS/PDMC*. Vol. 4346. LNCS. Springer, 2007.

[26] S. Fienblit. *Interleaving Review Technique*. 2003.

[27] T. Gee. *What to Look for in a CodeReview*. JetBrains Technical Series, 2016.

[28] J. Gimpel. "Software That Checks Software: The Impact of PC-lint". Em: *IEEE Software* 31.1 (2014).

[29] J. Gleick. *A Bug and a Crash by James Gleick*. https://around.com/ariane.html. 1996.

[30] K. Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Courier Corporation, 1992. ISBN: 978-0-486-66980-9.

[31] E. Goubault et al. "Static Analysis of the Accuracy in Control Systems: Principles and Experiments". Em: *FMICS*. Vol. 4916. LNCS. Springer, 2008.

[32] J. Graham-Cumming. *An Explanation of the Meltdown/Spectre Bugs for a Non-Technical Audience*. https://blog.cloudflare.com/meltdown-spectre-non-technical/. 2018.

[33] M. A. Hennell. "An experimental testbed for numerical software". Em: *The Computer Journal* 21.4 (1978).

[34] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". Em: *Commun. ACM* 12.10 (1969).

[35] G. J. Holzmann. "Static source code checking for user-defined properties". Em: *Integrated Design and Process Technology*. Vol. 2. 2002.

[36] G. J. Holzmann. "Cobra: Fast Structural Code Checking". Em: *SPIN*. ACM, 2017.

[37] *How Static Analysis Works*. http://www.verifysoft.com/en_grammatech_how_static_analysis_works.html.

[38] M. Llaguno. *2017 Coverity Scan Report*. Rel. téc. Synopsys, 2017.

[39] D. Marjamaki. *Cppcheck 1.81*. http://cppcheck.sourceforge.net/. 2017.

[40] R. C. Martin. "The Dependency Inversion Principle". Em: *C++ Report* 3 (1996).

[41] R. C. Martin. "The Interface Segregation Principle". Em: *C++ Report* 4 (1996).

[42] R. C. Martin. "The Liskov substitution principle". Em: *C++ Report* 2 (1996).

[43] R. C. Martin. "The Open-Closed Principle". Em: *C++ Report* 1 (1996).

[44] S. McConnell. *Code complete*. 2nd ed. Microsoft Press, 2004.

[45] A. Møller e M. I. Schwartzbach. *Static program analysis*. Department of Computer Science, Aarhus University, Denmark, 2012.

[46] T. Muske, R. Talluri e A. Serebrenik. "Repositioning of Static Analysis Alarms". Em: *27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: ACM, 2018, pp. 187–197. ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213850. URL: http://doi.acm.org/10.1145/3213846.3213850.

[47] I. Neamtiu, J. S. Foster e M. Hicks. "Understanding Source Code Evolution Using Abstract Syntax Tree Matching". Em: *SIGSOFT Softw. Eng. Notes* 30.4 (2005). URL: %5Curl%7Bhttp://doi.acm.org/10.1145/1082983.1083143%7D.

[48] N. Nethercote e J. Seward. "How to Shadow Every Byte of Memory Used by a Program". Em: *Proceedings of the 3rd International Conference on Virtual Execution Environments*. VEE '07. San Diego, California, USA: ACM, 2007, pp. 65–74. ISBN: 978-1-59593-630-1. DOI: 10.1145/1254810.1254820. URL: http://doi.acm.org/10.1145/1254810.1254820.

[49] T. Nicely. *Original Pentium FDIV flaw e-mail*. http://www.trnicely.net/pentbug/bugmail1.html.

[50] *Open-sourcing Facebook Infer: Identify bugs before you ship*. https://code.facebook.com/posts/1648953042007882/open-sourcing-facebook-infer-identify-bugs-before-you-ship/.

[51] *Parasoft C/C++test*. URL: https://www.parasoft.com/products/ctest.

[52] R. Patton. *Software testing*. Pearson Education India, 2006.

[53] C. Project. *Coverity Scan – Static Analysis*. https://scan.coverity.com/projects/211.

[54] L. Project. *Cppcheck - HTML report – LibreOffice 2018-01-27*. https://dev-builds.libreoffice.org/cppcheck_reports/master/index.html.

[55] PRQA Programming Research. *QA-C STATIC ANALYZER*. Rel. téc. http://www.prqa.com/static-analysis-software/qac-qacpp-static-analyzers/. 2017.

[56] *PVS-Studio Analyzer*. URL: https://www.viva64.com/en/pvs-studio/.

[57] P. Raatikainen. "Gödel's Incompleteness Theorems". Em: *The Stanford Encyclopedia of Philosophy*. 2015ª ed. Metaphysics Research Lab, Stanford University, 2015. URL: %5Curl%7Bhttps://plato.stanford.edu/archives/spr2015/entries/goedel-incompleteness/%7D.

[58] A. Raza, G. Vogel e E. Plödereder. "Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering". Em: *Ada-Europe 2006*. Vol. 4006. LNCS. Springer, 2006.

[59] J. Regehr. *A Guide to Undefined Behavior in C and C++, Part 1 – Embedded in Academia*. https://blog.regehr.org/archives/213. 2010.

[60] J. C. Reynolds. "Separation logic: a logic for shared mutable data structures". Em: *LICS*. IEEE, 2002.

[61] H. G. Rice. "Classes of recursively enumerable sets and their decision problems". Em: *Transactions of the American Mathematical Society* 74.2 (1953).

[62] *Rice's theorem*. http://kilby.stanford.edu/~rvg/154/handouts/Rice.html.

[63] K. Serebryany, D. Bruening, A. Potapenko e D. Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". Em: (2012), p. 10.

[64] P. Shved, M. Mandrykin e V. Mutilin. "Predicate Analysis with BLAST 2.7". Em: *TACAS*. Vol. 7214. LNCS. Springer, 2012.

[65] *Static and dynamic testing in the software development life cycle*. https://www.ibm.com/developerworks/library/se-static/.

[66] *Statistical Analysis of Floating Point Flaw: Intel White Paper*. Rel. téc. Intel, 2004.

[67] A. G. Stephenson et al. *Mars Climate Orbiter Mishap Investigation Board Phase I Report November 10, 1999*. Rel. téc. NASA, 1999.

[68] E. Teixeira, J. Antunes e N. Neves. "Avaliação de Ferramentas de Análise Estática de Código para Detecção de Vulnerabilidades". pt. Em: (2007), p. 17.

[69] *The LLVM Compiler Infrastructure Project*. URL: http://llvm.org/.

[70] R. Thomson. *British Airways reveals what went wrong with Terminal 5*. http://www.computerweekly.com/news/2240086013/British-Airways-reveals-what-went-wrong-with-Terminal-5.

[71] A. M. Turing. "On computable numbers, with an application to the Entscheidungsproblem". Em: *Proc. of the London Mathematical Society* (1937).

[72] Y. Tymchuk, M. Ghafari e O. Nierstrasz. "When QualityAssistant Meets Pharo: Enforced Code Critiques Motivate More Valuable Rules". en. Em: *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies - IWST'16*. Prague, Czech Republic: ACM Press, 2016, pp. 1–6. ISBN: 978-1-4503-4524-8. DOI: 10.1145/2991041.2991046. URL: http://dl.acm.org/citation.cfm?doid=2991041.2991046.

[73] Y. Tymchuk, M. Ghafari e O. Nierstrasz. "Renraku: The One Static Analysis Model to Rule Them All". Em: *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies*. IWST '17. Maribor, Slovenia: ACM, 2017, 13:1–13:10. ISBN: 978-1-4503-5554-4. DOI: 10.1145/3139903.3139919. URL: http://doi.acm.org/10.1145/3139903.3139919.

[74] *VisualCodeGrepper – Code security scanning tool*. URL: https://github.com/nccgroup/VCG.

[75] D. A. Wheeler. *Flawfinder*. Rel. téc. 2017.

[76] B. A. Wichmann et al. "Industrial perspective on static analysis". Em: *Software Engineering Journal* 10.2 (1995).

[77] K. Wissing. "Static Analysis of Dynamic Properties-Automatic Program Verification to Prove the Absence of Dynamic Runtime Errors." Em: *INFORMATIK*. Vol. 110. LNI. GI, 2007.

[78] Y. Xie e A. Aiken. "Saturn: A SAT-Based Tool for Bug Detection". Em: *CAV*. Vol. 3576. LNCS. Springer, 2005.

APÊNDICE A

Relatórios de resultados

Tabela A.2: Resultados da análise das 12 versões do Beanstalkd.

Erro		Versão		Reportado	Falso positivo	Ferramenta
Localização	Tipo	Introduzido	Corrigido			
beanstalkd.c:667	ML	0.3	0.4	✓	x	x
net.c:28	DS	0.3	1.4	x	x	CSA Infer
beanstalkd.c:41	RL	0.3	1.5	x	x	Infer
reserve.c:51	ND	0.3	1.5	x	x	Infer
prot.c:140	ND	0.3	1.5	x	x	Infer
beanstalkd.c:395	ND	0.3	0.5	x	x	Infer
beanstalkd.c:737	ML	0.3	0.5	x	x	Infer
beanstalkd.c:205	ML	0.4	0.5	✓	x	x
tube.c:60	ML	0.8	1.2	✓	x	x
ms.c:56	ML	0.8	x	x	✓	Infer
prot.c:320	ND	0.8	1.5	x	x	Infer
prot.c:374	ND	0.8	1.5	x	x	Infer
prot.c:946	DS	0.8	x	x	✓	Infer
prot.c:1134	DS	0.8	x	x	✓	Infer
prot.c:1170	DS	0.8	x	x	✓	Infer
prot.c:1230	DS	0.8	x	x	✓	Infer
prot.c:1388	DP	1.2	1.3	✓	x	x
job.c:170	ML	1.2	1.4	✓	x	x
job.c:48	DS	1.2	x	x	✓	Infer
tube.c:86	DS	1.2	x	x	✓	Infer
job.c:95	DS	1.2	x	x	✓	Infer
conn.c:224	DS	1.2	x	x	✓	Infer

Continua na página seguinte.

Tabela A.2 – *Continuação da página anterior.*

Erro		**Versão**		**Reportado**	**Falso positivo**	**Ferramenta**
Localização	Tipo	Introduzido	Corrigido			
prot.c:1493	ML	1.2	1.4	✓	x	x
prot.c:1583	ND	1.2	x	x	x	Infer
cut.c:222	RL	1.2	1.5	x	x	Cppcheck
binlog.c:176	BO	1.4.1	1.4.2	✓	x	x
binlog.c:215	ND	1.4.1	1.4.2	x	x	CSA
binlog.c:723	ND	1.4.1	1.5	x	x	Infer
job.c:70	ML	1.4.2	1.4.3	✓	x	Predator
prot.c:974	ML	1.4.2	1.4.3	✓	x	x
prot.c:1017	ML	1.4.3	1.4.4	✓	x	x
prot.c:1736	SegF	1.5	1.6	✓	x	x
net.c:31	DS	1.5	1.9	x	x	CSA Infer
heap.c:98	ML	1.5	x	x	✓	Infer
walg.c:293	ML	1.5	x	x	✓	Infer
walg.c:293	RL	1.5	x	x	✓	Infer
walg.c:357	ML	1.5	x	x	✓	Infer
walg.c:357	RL	1.5	x	x	✓	Infer
walg.c:426	RL	1.5	x	x	✓	Infer
prot.c:514	ND	1.5	1.8	x	x	Infer
prot.c:554	ND	1.5	1.8	x	x	Infer
conn.c:232	SegF	1.6	1.7	✓	x	x
file.c:204	ND	1.6	1.7	✓	x	CSA
file.c:325	ND	1.6	1.7	✓	x	CSA
prot.c:501	ND	1.6	x	x	x	Infer
conn.c:244	InvF	1.8	1.9	✓	x	x
integ-test.c:343	InvF	1.8	1.9	✓	x	x
prot.c:710	InvF	1.8	1.9	✓	x	x
darwin.c:84	DS	1.8	x	x	x	Infer
walg.c:416	DS	1.9	1.10	✓	x	CSA Infer
testheap.c:222	ML	1.10	x	x	x	Cppcheck Predator

Tabela A.3: Resultados da análise da versão 2.7 do Tmux.

Erro		**Versão**		**Reportado**	**Falso positivo**	**Ferramenta**
Localização	Tipo	Introduzido	Corrigido			
window.c:57	ND	1.1	x	x	✓	CSA Infer

Continua na página seguinte.

Tabela A.3 – *Continuação da página anterior.*

Erro		Versão		Reportado	Falso positivo	Ferramenta
Localização	Tipo	Introduzido	Corrigido			
cmd-respawn-window.c:73	ND	1.1	x	x	x	CSA
imsgbuffer.c:217	UAF	1.1	x	x	✓	CSA
environ.c:30	ND	1.1	x	x	✓	CSA Infer
options.c:134	ND	1.1	x	x	✓	CSA
session:38	ND	1.4	x	x	✓	CSA Infer
key-bindings.c:27	ND	1.6	x	x	✓	CSA Infer
screen.c:91	ND	1.7	x	x	✓	CSA Infer
server-client.c:725	UV	1.7	x	x	✓	Infer
cmd-run-shell.c:119	ML	1.8	x	x	✓	Infer
arguments.c:215	ML	1.9a	x	x	✓	Infer
cmd-string.c:282	ND	1.9a	x	x	✓	Infer
options.c:194	ND	1.9a	x	x	✓	CSA
paste.c:287	ND	1.9a	x	x	✓	Infer
status.c:1208	ND	1.9a	x	x	✓	Infer
status.c:1285	ND	1.9a	x	x	✓	Infer
utf8.c:239	ND	1.9a	x	x	✓	Infer
utf8.c:259	ND	1.9a	x	x	✓	Infer
utf8.c:279	ND	1.9a	x	x	✓	Infer
utf8.c:294	ND	1.9a	x	x	✓	Infer
window.c:1283	ND	1.9a	x	x	✓	Infer
window.c:1329	ND	1.9a	x	x	✓	Infer
window.c:1375	ND	1.9a	x	x	✓	Infer
key-bindings.c:28	ND	2.0	x	x	✓	CSA Infer
compat/imsg.c:90	DS	2.0	x	x	✓	Infer
window.c:668	UAF	2.0	x	x	x	CSA
key-bindings.c:108	UAF	2.0	x	x	x	Cppcheck
cmd-string.c:235	ND	2.3	x	x	✓	Infer
cmd-string.c:239	ND	2.3	x	x	✓	Infer
cmd-find.c:202	ND	2.3	x	x	✓	Infer
hooks.c:33	ND	2.3	x	x	✓	CSA Infer
key-bindings.c:70	ML	2.3	x	x	✓	Infer
arguments.c:41	ND	2.4	x	x	✓	CSA Infer
cmd-queue.c:189	ML	2.4	x	x	x	Infer
cmd-queue.c:304	ND	2.4	x	x	✓	CSA
cmd-wait-for.c:63	ND	2.4	x	x	✓	CSA Infer
format.c:101	ND	2.4	x	x	✓	CSA Infer

Continua na página seguinte.

Tabela A.3 – *Continuação da página anterior.*

Erro		**Versão**		**Reportado**	**Falso positivo**	**Ferramenta**
Localização	Tipo	Introduzido	Corrigido			
format.c:141	ND	2.4	x	x	✓	CSA Infer
mode-tree.c:710	ML	2.4	x	x	✓	Infer
options.c:98	ND	2.4	x	x	✓	CSA
options.c:80	ND	2.4	x	x	✓	CSA Infer
paste.c:53	ND	2.4	x	x	✓	CSA Infer
paste.c:57	ND	2.4	x	x	✓	CSA Infer
session.c:55	ND	2.4	x	x	✓	CSA Infer
server-client.c:381	DS	2.4	x	x	✓	Infer
window.c:77	ND	2.4	x	x	✓	CSA Infer
window.c:78	ND	2.4	x	x	✓	CSA Infer
window.c:79	ND	2.4	x	x	✓	CSA Infer
grid.c:1014	OB	2.7	x	✓	x	x
window-copy.c:2446	ND	2.7	x	✓	x	x
arguments.c:100	UAF	2.7	x	x	✓	Infer
cmd.c:237	ML	2.7	x	x	✓	Infer
cmd-capture-pane.c:102	ML	2.7	x	x	✓	Infer
cmd-capture-pane.c:98	ML	2.7	x	x	✓	Infer
cmd-command-prompt.c:151	DS	2.7	x	x	✓	Infer
cmd-display-panes.c:78	ML	2.7	x	x	✓	Infer
compat/daemon.c:72	RL	2.7	x	x	✓	Infer
compat/imsg.c:264	UV	2.7	x	x	✓	Cppcheck
format.c:199	UAF	2.7	x	x	✓	Infer
format.c:229	DS	2.7	x	x	✓	Infer
format.c:798	DS	2.7	x	x	✓	Infer
layout.c:436	UAF	2.7	x	x	✓	Infer
layout.c:83	UAF	2.7	x	x	x	CSA
layout-custom.c:158	ND	2.7	x	x	✓	Infer
osdep-freebsd.c:173	OB	2.7	x	x	✓	Cppcheck
screen.c:49	UAF	2.7	x	x	✓	CSA Infer
screen-redraw.c:454	DS	2.7	x	x	✓	Infer
session.c:644	DS	2.7	x	x	✓	Infer
status.c:1041	DS	2.7	x	x	✓	Infer
tty-term.c:323	ML	2.7	x	x	✓	Infer
window.c:429	ML	2.7	x	x	✓	Infer
window-copy.c:1298	DS	2.7	x	x	✓	Infer

Tabela A.1: Resultados da análise das 2 versões do SDS.

Erro		Versão		Reportado	Falso positivo	Ferramenta
Localização	Tipo	Introduzido	Corrigido			
sds.c:303	ML	1	2	✓	x	x
sds.c:470	ML	1	x	✓	✓	x
sds.c:882	ML	1	2	✓	x	x
sds.c:881	ML	1	2	✓	x	x
sds.c:159	ML	1	2	✓	x	Cppcheck
sds.c:160	ND	1	2	✓	x	Infer
sds.c:717	ND	1	x	✓	x	x
sds.c:817	ND	1	x	x	x	Infer
sds.c:891	ND	1	2	x	x	CSA
sds.c:894	ND	1	2	x	x	CSA
sds.c:63	InvD	1	x	x	✓	Predator
sds.c:147	InvD	1	x	x	✓	Predator
sds.c:793	ML	1	x	x	✓	Predator
sds.c:92	ND	2	x	✓	x	Infer
sds.c:1109	ND	2	x	x	x	Infer
sds.c:580	DS	2	x	x	✓	Infer
sds.c:136	InvD	2	x	x	✓	Predator
sds.c:98	InvD	2	x	x	✓	Predator
sds.c:96	InvD	2	x	x	✓	Predator
sds.c:94	InvD	2	x	x	✓	Predator
sds.c:92	InvD	2	x	x	✓	Predator
sds.c:161	InvF	2	x	x	✓	CSA Predator
sds.c:161	ML	2	x	x	✓	Predator
sds.c:1121	ML	2	x	x	✓	Predator
sds.h:87	ND	2	x	x	✓	Predator
sds.h:197	InvD	2	x	x	✓	Predator
sds.c:229	InvF	2	x	x	✓	CSA
sds.c:1240	DS	2	x	x	x	CSA Infer

Tabela A.4: Resultados da análise da versão 3.24.0 do SQLite.

Erro		Versão		Reportado	Falso positivo	Ferramenta
Localização	Tipo	Introduzido	Corrigido			
shell.c:10203	DS	3.21.0	x	x	x	CSA Infer
shell.c:10978	RL	3.21.0	x	x	✓	Cppcheck Infer
shell.c:11873	UV	3.21.0	x	x	✓	Infer
shell.c:11874	UV	3.21.0	x	x	✓	Infer
shell.c:11875	UV	3.21.0	x	x	✓	Infer
shell.c:12365	RL	3.21.0	x	x	✓	Cppcheck

Continua na página seguinte.

Tabela A.4 – *Continuação da página anterior.*

Erro		**Versão**		**Reportado**	**Falso positivo**	**Ferramenta**
Localização	Tipo	Introduzido	Corrigido			
shell.c:13799	DS	3.21.0	x	x	x	CSA Infer
shell.c:14022	DS	3.21.0	x	x	✓	Infer
shell.c:14953	DS	3.21.0	x	x	x	CSA Infer
shell.c:3329	DS	3.21.0	x	x	✓	Infer
shell.c:585	ML	3.21.0	x	x	x	Cppcheck CSA Infer
sqlite3.c:101332	UV	3.21.0	x	x	✓	Infer
sqlite3.c:101333	UV	3.21.0	x	x	✓	Infer
sqlite3.c:102072	DS	3.21.0	x	x	✓	Infer
sqlite3.c:102497	ND	3.21.0	x	x	✓	Cppcheck
sqlite3.c:102504	ND	3.21.0	x	x	✓	Cppcheck
sqlite3.c:103813	ND	3.21.0	x	x	✓	CSA
sqlite3.c:104022	ND	3.21.0	x	x	✓	Infer
sqlite3.c:104629	ND	3.21.0	x	x	x	Infer
sqlite3.c:104696	ND	3.21.0	x	x	✓	Cppcheck
sqlite3.c:106296	ND	3.21.0	x	x	x	Infer
sqlite3.c:106912	ND	3.21.0	x	x	✓	Cppcheck
sqlite3.c:107329	ND	3.21.0	x	x	✓	CSA
sqlite3.c:108114	ND	3.21.0	x	x	✓	Infer
sqlite3.c:108478	ND	3.21.0	x	x	✓	Infer
sqlite3.c:109158	DS	3.21.0	x	x	✓	Infer
sqlite3.c:112344	ND	3.21.0	x	x	x	Infer
sqlite3.c:113115	ND	3.21.0	x	x	✓	CSA
sqlite3.c:113119	ND	3.21.0	x	x	✓	CSA
sqlite3.c:113253	ND	3.21.0	x	x	✓	Infer
sqlite3.c:113253	ND	3.21.0	x	x	x	Infer
sqlite3.c:113375	ND	3.21.0	x	x	x	Infer
sqlite3.c:114558	ND	3.21.0	x	x	✓	Infer
sqlite3.c:115133	ND	3.21.0	x	x	✓	Infer
sqlite3.c:115601	ND	3.21.0	x	x	✓	CSA
sqlite3.c:119263	DS	3.21.0	x	x	✓	CSA Infer
sqlite3.c:120122	DS	3.21.0	x	x	x	CSA Infer
sqlite3.c:120609	UV	3.21.0	x	x	✓	Infer
sqlite3.c:120643	UV	3.21.0	x	x	✓	Infer
sqlite3.c:120658	UV	3.21.0	x	x	✓	Infer

Continua na página seguinte.

Tabela A.4 – *Continuação da página anterior.*

Erro		**Versão**		**Reportado**	**Falso positivo**	**Ferramenta**
Localização	Tipo	Introduzido	Corrigido			
sqlite3.c:121000	UV	3.21.0	x	x	✓	Infer
sqlite3.c:121032	UV	3.21.0	x	x	✓	Infer
sqlite3.c:121044	UV	3.21.0	x	x	✓	Infer
sqlite3.c:122321	ND	3.21.0	x	x	✓	Cppcheck
sqlite3.c:123485	ND	3.21.0	x	x	✓	Infer
sqlite3.c:123495	ND	3.21.0	x	x	✓	Infer
sqlite3.c:123627	DS	3.21.0	x	x	✓	Infer
sqlite3.c:125249	ND	3.21.0	x	x	✓	CSA
sqlite3.c:127248	UV	3.21.0	x	x	✓	Infer
sqlite3.c:127301	UV	3.21.0	x	x	✓	Infer
sqlite3.c:128851	ND	3.21.0	x	x	x	Infer
sqlite3.c:130783	ML	3.21.0	x	x	✓	Infer
sqlite3.c:133758	DS	3.21.0	x	x	✓	Infer
sqlite3.c:133957	DS	3.21.0	x	x	✓	Infer
sqlite3.c:136641	ND	3.21.0	x	x	✓	Cppcheck
sqlite3.c:146676	ND	3.21.0	x	x	✓	Infer
sqlite3.c:149560	DS	3.21.0	x	x	✓	Infer
sqlite3.c:150066	ND	3.21.0	x	x	✓	Infer
sqlite3.c:150805	ND	3.21.0	x	x	✓	CSA
sqlite3.c:153100	DS	3.21.0	x	x	✓	Infer
sqlite3.c:153934	DS	3.21.0	x	x	✓	Infer
sqlite3.c:154163	ND	3.21.0	x	x	✓	CSA
sqlite3.c:154548	ND	3.21.0	x	x	✓	CSA
sqlite3.c:155513	DS	3.21.0	x	x	✓	Infer
sqlite3.c:157228	ND	3.21.0	x	x	✓	CSA
sqlite3.c:157619	ND	3.21.0	x	x	✓	CSA
sqlite3.c:158457	DS	3.21.0	x	x	✓	Infer
sqlite3.c:158649	DS	3.21.0	x	x	✓	Infer
sqlite3.c:159638	ND	3.21.0	x	x	✓	CSA
sqlite3.c:159686	ND	3.21.0	x	x	✓	CSA
sqlite3.c:159885	ML	3.21.0	x	x	✓	Infer
sqlite3.c:161317	DS	3.21.0	x	x	✓	Infer
sqlite3.c:166275	ND	3.21.0	x	x	✓	CSA
sqlite3.c:167205	DS	3.21.0	x	x	✓	Infer
sqlite3.c:169136	ND	3.21.0	x	x	✓	CSA
sqlite3.c:170953	ND	3.21.0	x	x	✓	CSA
sqlite3.c:171730	UV	3.21.0	x	x	✓	Infer
sqlite3.c:171730	UV	3.21.0	x	x	✓	Infer
sqlite3.c:171738	UV	3.21.0	x	x	✓	Infer
sqlite3.c:171738	UV	3.21.0	x	x	✓	Infer
sqlite3.c:171767	UV	3.21.0	x	x	✓	Infer
sqlite3.c:171767	UV	3.21.0	x	x	✓	Infer

Continua na página seguinte.

Tabela A.4 – *Continuação da página anterior.*

Erro		**Versão**		**Reportado**	**Falso positivo**	**Ferramenta**
Localização	Tipo	Introduzido	Corrigido			
sqlite3.c:171789	ND	3.21.0	x	x	✓	CSA
sqlite3.c:172125	ND	3.21.0	x	x	✓	Infer
sqlite3.c:172144	ND	3.21.0	x	x	✓	Infer
sqlite3.c:172146	ND	3.21.0	x	x	✓	Infer
sqlite3.c:173091	ND	3.21.0	x	x	✓	Infer
sqlite3.c:26792	DS	3.21.0	x	x	✓	Infer
sqlite3.c:29304	DS	3.21.0	x	x	✓	Infer
sqlite3.c:29309	DS	3.21.0	x	x	✓	Infer
sqlite3.c:29680	UV	3.21.0	x	x	✓	Infer
sqlite3.c:32349	UV	3.21.0	x	x	✓	Infer
sqlite3.c:32831	DS	3.21.0	x	x	✓	CSA Infer
sqlite3.c:35249	DS	3.21.0	x	x	✓	Infer
sqlite3.c:47476	UV	3.21.0	x	x	✓	Infer
sqlite3.c:47514	UV	3.21.0	x	x	✓	Infer
sqlite3.c:47517	UV	3.21.0	x	x	✓	Infer
sqlite3.c:49151	ML	3.21.0	x	x	✓	Infer
sqlite3.c:49158	ML	3.21.0	x	x	✓	Infer
sqlite3.c:49163	UV	3.21.0	x	x	✓	Infer
sqlite3.c:49185	UV	3.21.0	x	x	✓	Infer
sqlite3.c:49188	UV	3.21.0	x	x	✓	Infer
sqlite3.c:52707	ML	3.21.0	x	x	✓	Infer
sqlite3.c:52750	ML	3.21.0	x	x	✓	Infer
sqlite3.c:54219	ML	3.21.0	x	x	✓	Infer
sqlite3.c:55959	DS	3.21.0	x	x	✓	CSA
sqlite3.c:57100	ND	3.21.0	x	x	✓	Cppcheck
sqlite3.c:60444	ND	3.21.0	x	x	✓	CSA
sqlite3.c:60678	ND	3.21.0	x	x	✓	Infer
sqlite3.c:64483	UV	3.21.0	x	x	✓	Infer
sqlite3.c:67330	DS	3.21.0	x	x	x	CSA
sqlite3.c:67508	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69432	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69436	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69492	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69543	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69548	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69564	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69573	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69588	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69606	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69607	UV	3.21.0	x	x	✓	Infer

Continua na página seguinte.

Tabela A.4 – *Continuação da página anterior.*

Erro		**Versão**		**Reportado**	**Falso positivo**	**Ferramenta**
Localização	Tipo	Introduzido	Corrigido			
sqlite3.c:69630	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69657	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69715	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69721	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69767	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69777	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69777	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69782	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69812	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69813	UV	3.21.0	x	x	✓	Infer
sqlite3.c:69903	UV	3.21.0	x	x	✓	Infer
sqlite3.c:72545	DS	3.21.0	x	x	✓	Infer
sqlite3.c:73233	UV	3.21.0	x	x	✓	CSA
sqlite3.c:73543	ND	3.21.0	x	x	✓	CSA
sqlite3.c:74808	ND	3.21.0	x	x	✓	CSA
sqlite3.c:74999	ND	3.21.0	x	x	✓	CSA
sqlite3.c:75007	ND	3.21.0	x	x	✓	CSA
sqlite3.c:75200	ND	3.21.0	x	x	✓	CSA
sqlite3.c:76677	UV	3.21.0	x	x	✓	Infer
sqlite3.c:83294	DS	3.21.0	x	x	✓	CSA Infer
sqlite3.c:85387	DS	3.21.0	x	x	✓	Infer
sqlite3.c:85391	DS	3.21.0	x	x	✓	Infer
sqlite3.c:85406	DS	3.21.0	x	x	✓	Infer
sqlite3.c:88659	DS	3.21.0	x	x	✓	CSA
sqlite3.c:88747	DS	3.21.0	x	x	✓	CSA
sqlite3.c:90774	UV	3.21.0	x	x	✓	Cppcheck
sqlite3.c:90784	UV	3.21.0	x	x	✓	Cppcheck
sqlite3.c:91541	ND	3.21.0	x	x	✓	CSA Infer
sqlite3.c:91920	ND	3.21.0	x	x	x	CSA
sqlite3.c:92342	ND	3.21.0	x	x	✓	CSA
sqlite3.c:92692	ND	3.21.0	x	x	✓	CSA
sqlite3.c:93841	ML	3.21.0	x	x	✓	Infer
sqlite3.c:95530	ND	3.21.0	x	x	✓	CSA
shell.c:2759	UV	3.22.0	x	x	✓	Infer
shell.c:7807	DS	3.22.0	x	x	✓	Infer
sqlite3.c:132810	ND	3.22.0	x	x	x	Infer
sqlite3.c:21733	ND	3.22.0	x	x	✓	CSA
sqlite3.c:76633	ND	3.22.0	x	x	✓	Cppcheck CSA

Continua na página seguinte.

Tabela A.4 – *Continuação da página anterior.*

Erro		**Versão**		**Reportado**	**Falso positivo**	**Ferramenta**
Localização	Tipo	Introduzido	Corrigido			
shell.c:4570	ND	3.23.1	x	x	✓	CSA
shell.c:4877	DS	3.23.1	x	x	x	CSA Infer
shell.c:5241	UV	3.23.1	x	x	✓	CSA
shell.c:5576	ND	3.23.1	x	x	✓	Infer
sqlite3.c:112108	ND	3.24.0	x	x	x	Infer
sqlite3.c:114841	UV	3.24.0	x	x	✓	Infer
sqlite3.c:114844	UV	3.24.0	x	x	✓	Infer
sqlite3.c:130155	ND	3.24.0	x	x	x	CSA
sqlite3.c:130168	ND	3.24.0	x	x	✓	Infer
sqlite3.c:135444	DS	3.24.0	x	x	x	CSA Infer
sqlite3.c:174093	DS	3.24.0	x	x	✓	Infer
sqlite3.c:181388	ND	3.24.0	x	x	✓	Infer
sqlite3.c:195659	ND	3.24.0	x	x	✓	Infer
sqlite3.c:196716	UV	3.24.0	x	x	✓	Infer
sqlite3.c:197310	DS	3.24.0	x	x	✓	Infer
sqlite3.c:199054	ND	3.24.0	x	x	x	CSA
sqlite3.c:200165	ND	3.24.0	x	x	✓	CSA
sqlite3.c:201114	ND	3.24.0	x	x	✓	CSA
sqlite3.c:201953	ND	3.24.0	x	x	✓	CSA
sqlite3.c:202027	UV	3.24.0	x	x	✓	Infer
sqlite3.c:203408	ND	3.24.0	x	x	✓	Infer
sqlite3.c:203411	ND	3.24.0	x	x	✓	Infer
sqlite3.c:203453	DS	3.24.0	x	x	✓	Infer
sqlite3.c:203457	DS	3.24.0	x	x	✓	Infer
sqlite3.c:203552	UV	3.24.0	x	x	✓	Infer
sqlite3.c:203553	UV	3.24.0	x	x	✓	Infer
sqlite3.c:204864	DS	3.24.0	x	x	✓	CSA Infer
sqlite3.c:204873	UV	3.24.0	x	x	✓	Infer
sqlite3.c:206125	DS	3.24.0	x	x	✓	Infer
sqlite3.c:207193	ND	3.24.0	x	x	✓	Infer
sqlite3.c:207263	UV	3.24.0	x	x	✓	Infer
sqlite3.c:207528	ND	3.24.0	x	x	✓	CSA
sqlite3.c:208661	UV	3.24.0	x	x	✓	Infer
sqlite3.c:210473	ND	3.24.0	x	x	✓	CSA
sqlite3.c:210486	ND	3.24.0	x	x	✓	CSA
sqlite3.c:210501	ND	3.24.0	x	x	✓	CSA
sqlite3.c:210519	ND	3.24.0	x	x	✓	CSA
sqlite3.c:211076	DS	3.24.0	x	x	✓	Infer

Continua na página seguinte.

Tabela A.4 – *Continuação da página anterior.*

Erro		**Versão**		**Reportado**	**Falso positivo**	**Ferramenta**
Localização	Tipo	Introduzido	Corrigido			
sqlite3.c:211087	DS	3.24.0	x	x	✓	Infer
sqlite3.c:211390	DS	3.24.0	x	x	✓	Infer
sqlite3.c:75150	DS	3.24.0	x	x	x	CSA Infer
sqlite3.c:76634	ND	3.24.0	x	x	✓	Cppcheck
sqlite3.c:76636	ND	3.24.0	x	x	✓	Cppcheck
sqlite3.c:76658	ND	3.24.0	x	x	✓	Cppcheck
sqlite3.c:76659	ND	3.24.0	x	x	✓	Cppcheck

Apêndice B

Exemplos mínimos

Este apêndice está organizado por tipo de erro, contendo cada secção listagens com o código dos exemplos mínimos e os relatórios das ferramentas utilizadas.

B.1 Dead store

A Listagem B.1 corresponde ao exemplo mínimo de um falso positivo identificado pelo Infer (Listagem B.2). Este erro é considerado um falso positivo, porque o endereço é inicializado com o valor `NULL` e, portanto o valor do mesmo só é utilizado depois de lhe ser atribuído um novo valor. Caso o endereço tivesse sido inicializado com o valor 0, o Infer iria devolver o mesmo tipo de resultado. Quando este erro surge com este tipo de padrão não é reportado por mais nenhuma das restantes ferramentas. Na Listagem B.3 o erro é corrigido e o Infer não devolve qualquer tipo de resultado. Por fim, na Listagem B.4 está representado um exemplo de um erro do mesmo tipo, inicializado com um valor diferente de `NULL` ou 0, que é reportado por todas as ferramentas capazes de identificar erros do tipo *dead store* (Listagem B.5 e Figura B.1).

Listagem B.1: Exemplo mínimo de um erro *dead store* classificado como falso positivo.

```
1  #include <stdio.h>
2
3  int main () {
4    int var = 20;
5    int *ptr_a = NULL;
6
7    ptr_a = &var;
8    printf("Value:%d\n",*ptr_a);
9
10     return 0;
```

```
11 }
```

Listagem B.2: Relatório de resultados d0 Infer para o exemplo mínimo da Listagem B.1.

```
1  Found 1 issue
2
3  dead_store_false_positive_infer.c:7: error: DEAD_STORE
4    The value written to &ptr_a is never used.
5    5.   int main () {
6    6.     int var = 20;
7    7. >   int *ptr_a = NULL;
8    8.
9    9.     ptr_a = &var;
10
11 Summary of the reports
12
13   DEAD_STORE: 1
```

Listagem B.3: Exemplo mínimo de um erro do tipo *dead store* corrigido.

```
1  #include <stdio.h>
2
3  int main () {
4    int var = 20;
5    int *ptr_a = &var;
6    printf("Value:%d\n",*ptr_a);
7
8      return 0;
9  }
```

Listagem B.4: Exemplo mínimo de um erro do tipo *dead store* classificado como verdadeiro positivo.

```
1  #include <stdio.h>
2
3  int main () {
4    int a = 20;
5    int b = 10;
6    int *ptr_a = &a;
7
8    ptr_a = &b;
9    printf("Value:%d\n",*ptr_a);
10
11     return 0;
12 }
```

Listagem B.5: Relatório de resultados da ferramenta Infer para o exemplo mínimo da Listagem B.4.

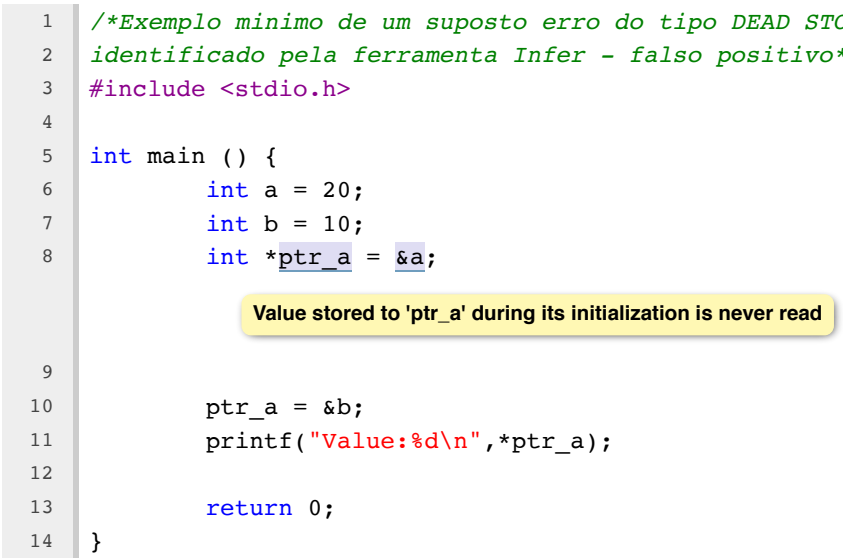
```
Found 1 issue

dead_store_false_positive_infer.c:8: error: DEAD_STORE
  The value written to &ptr_a is never used.
  6.     int a = 20;
  7.     int b = 10;
  8. >   int *ptr_a = &a;
  9.
  10.      ptr_a = &b;

Summary of the reports

  DEAD_STORE: 1
```

Figura B.1: Relatório de erros devolvido pela ferramenta Clang Static Analyzer para o exemplo mínimo da Listagem B.4.



B.2 Fuga de memória

B.2.1 Argumento passado ao malloc

Na Listagem B.6 está representado o exemplo mínimo onde o argumento passado à função `malloc` é do tipo `ptr*` e, portanto, o Infer (até à versão 0.13.1) não é capaz de identificar a fuga de memória presente no código. Por outro, o Predator e o Clang Static Analyzer conseguem identificar o erro, como pode ser observado no relatório de resultados da Listagem B.7 e da Figura B.2. Na Listagem B.8 a fuga de memória foi corrigida e não é reportada por nenhuma das ferramentas. De notar que, qualquer erro que exista no

código depois da chamada à função `malloc`, passando o argumento `ptr*`, é ignorado pelo Infer. Na Listagem B.9 o argumento passado à função `malloc` é do mesmo tipo da estrutura utilizada, logo, qualquer versão do Infer é capaz de identificar a fuga de memória presente no código (Listagem B.10). Além disso, o Predator e o Clang Static Analyzer (Listagem B.11 e Figura B.3) também conseguem identificar o erro, tal como já acontecia no exemplo mínimo anterior.

Listagem B.6: Exemplo mínimo de um erro do tipo fuga de memória.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct st{
   int value;
}st;

st * create(int value){
   st *new = malloc(sizeof(*new));
   if(new == NULL) return NULL;
   new -> value = value;
   return new;
}

int main(){
    st *new = create(5);
    printf("Value:%d\n",new->value);
    return 0;
}
```

Listagem B.7: Relatório de resultados da ferramenta Predator para o exemplo mínimo da Listagem B.6.

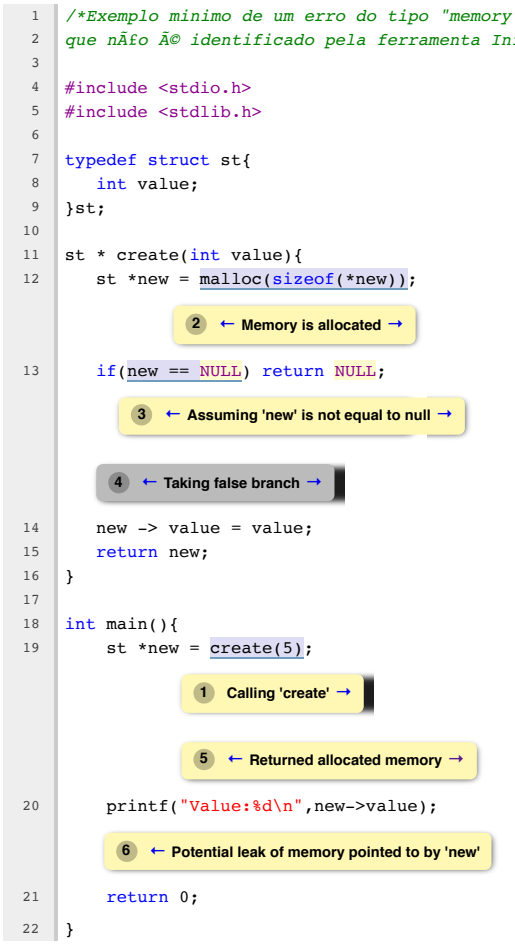
```
memory_leak_false_negative_infer.c:20:11: warning: memory
leak detected while destroying a variable on stack
[-fplugin=libsl.so]
/home/fct/predator/cl/cl_easy.cc:83: note: clEasyRun()
took 0.001 s [internal location] [-fplugin=libsl.so]
```

Listagem B.8: Exemplo mínimo de um erro do tipo fuga de memória corrigido.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct st{
   int value;
}st;
st * create(int value){
   st *new = malloc(sizeof(st));
```

Figura B.2: Relatório de erros devolvido pela ferramenta Clang Static Analyzer para o exemplo mínimo da Listagem B.6.




```
9     if(new == NULL) return NULL;
10    new -> value = value;
11    return new;
12 }
13 int main(){
14      st *new = create(5);
15      free(new);
16      return 0;
17 }
```

Listagem B.9: Exemplo mínimo de um erro do tipo fuga de memória identificado pelo Infer.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct st{
5     int value;
```

```
6  }st;
7
8  st * create(int value){
9     st *new = malloc(sizeof(st));
10    if(new == NULL) return NULL;
11    new -> value = value;
12    return new;
13 }
14
15 int main(){
16     st *new = create(5);
17   if(new){
18     printf("Value:%d\n",new->value);
19   }
20     return 0;
21 }
```

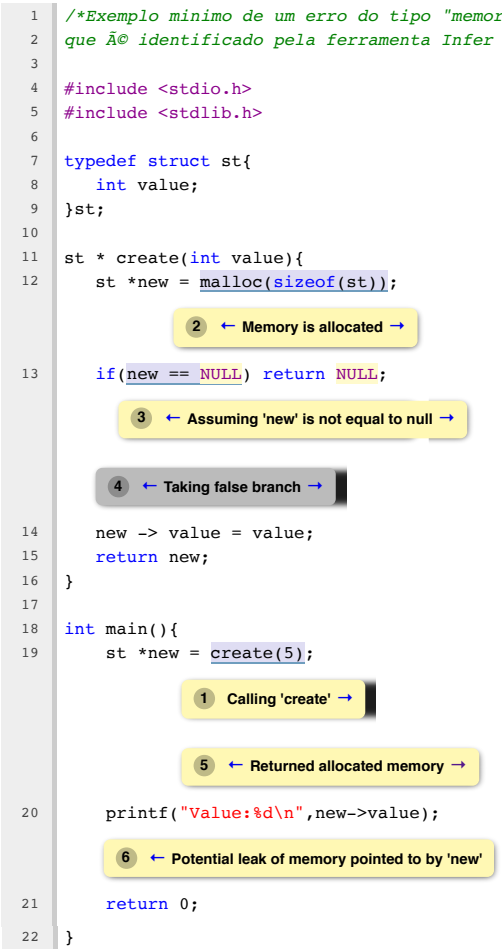
Listagem B.10: Relatório de resultados da ferramenta Infer para o exemplo mínimo da Listagem B.9.

```
1  Found 1 issue
2
3  memory_leak_true_positive_infer.c:21: error: MEMORY_LEAK
4    memory dynamically allocated to `new` by call to
5    `create()` at line 19, column 15 is not reachable
6    after line 21, column 5.
7    19.        st *new = create(5);
8    20.        if(new){
9    21. >      printf("Value:%d\n",new->value);
10   22.        }
11   23.        return 0;
12
13 Summary of the reports
14
15   MEMORY_LEAK: 1
```

Listagem B.11: Relatório de resultados da ferramenta Predator para o exemplo mínimo da Listagem B.9.

```
1  memory_leak_true_positive_infer.c:20:11: warning: memory
2  leak detected while destroying a variable on stack
3  [-fplugin=libsl.so]
4  /home/fct/predator/cl/cl_easy.cc:83: note: clEasyRun()
5  took 0.001 s [internal location] [-fplugin=libsl.so]
```

Figura B.3: Relatório de erros devolvido pela ferramenta Clang Static Analyzer para o exemplo mínimo da Listagem B.9.



B.2.2 Estrutura com apontadores

Na Listagem B.12 está representado o exemplo mínimo de um erro do tipo fuga de memória que é identificado por todas as ferramentas (consultar a Listagem B.13, a Listagem B.14 e a Listagem B.15), exceto pelo Clang Static Analyzer. Na Listagem B.16 a fuga de memória foi corrigida e, portanto, não é reportada por nenhuma das ferramentas.

Listagem B.12: Exemplo mínimo de um erro do tipo fuga de memória numa estrutura com apontadores.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct st{
   int *value;
}st;

int main(){
    st *new = malloc(sizeof(st));
```

```
10    if(!new) return -1;
11    new->value = malloc(sizeof(int));
12    free(new);
13    return 0;
14  }
```

Listagem B.13: Relatório de erros devolvido pela ferramenta Cppcheck para o exemplo mínimo da Listagem B.12.

```
1  Checking memory_leak_true_positive_structwithpointer.c ...
2  [memory_leak_true_positive_structwithpointer_infer.c:19]:
3  (error) Memory leak: new.value
```

Listagem B.14: Relatório de erros devolvido pela ferramenta Infer para o exemplo mínimo da Listagem B.12.

```
1  Found 1 issue
2
3  memory_leak_true_positive_structwithpointer.c:19:
4  error: MEMORY_LEAK memory dynamically allocated to
5  `new->value` by call to `malloc()` at line 14,
6  column 15 is not reachable after line 19, column 2.
7    17.       return -1;
8    18.     }*/
9    19. >   free(new);
10   20.     return 0;
11
12 Summary of the reports
13
14   MEMORY_LEAK: 1
```

Listagem B.15: Relatório de erros devolvido pela ferramenta Predator para o exemplo mínimo da Listagem B.12.

```
1  memory_leak_true_positive_structwithpointer.c:19:6:
2  warning: memory leak detected while destroying a
3  heap object [-fplugin=libsl.so]
4  /home/fct/predator/cl/cl_easy.cc:83: note: clEasyRun()
5  took 0.001 s [internal location] [-fplugin=libsl.so]
```

Listagem B.16: Exemplo mínimo de um erro do tipo fuga de memória numa estrutura com apontadores corrigido.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
```

```
4  typedef struct st{
5     int *value;
6  }st;
7
8  int main(){
9      st *new = malloc(sizeof(st));
10   if(!new) return -1;
11   new->value = malloc(sizeof(int));
12   free(new->value);
13   free(new);
14   return 0;
15 }
```

B.2.3 Falta de verificação do realloc

A falta de verificação de uma operação de memória é um tipo de erro apenas identificado pelo Infer. No entanto, quando se trata da verificação de uma chamada à função `realloc` (Listagem B.17) o Cppcheck identifica um erro do tipo fuga de memória, no caso do exemplo mínimo apresentado nesta secção o Infer também identifica um erro deste tipo. Os relatórios de resultados do Infer e do Cppcheck encontram-se na Listagem B.18 e na Listagem B.19, respetivamente. O Clang Static Analyzer não reporta qualquer resultado para este exemplo. O Predator devolve um falso positivo, uma vez que esta ferramenta ignora a chamada à função `realloc`. Na Listagem B.20 encontra-se representada a correção da fuga de memória do exemplo anterior, sendo que não é reportado qualquer tipo de erro pelas ferramentas, com a exceção do Predator que, como já foi referido, reporta um falso positivo.

Listagem B.17: Exemplo mínimo de um erro do tipo fuga de memória utilizando uma função `realloc`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct st{
5     int value;
6  }st;
7
8  int main(){
9      st *new = malloc(sizeof(st));
10   if(!new) return -1;
11   new = realloc(new,2*sizeof(st));
12   free(new);
13   return 0;
14 }
```

Listagem B.18: Relatório de erros devolvido pela ferramenta Cppcheck para o exemplo mínimo da Listagem B.17.

```
Checking memory_leak_true_positive_realloc.c ...
[memory_leak_true_positive_realloc_infer.c:14]:
(error) Common realloc mistake: 'new' nulled
but not freed upon failure
```

Listagem B.19: Relatório de erros devolvido pela ferramenta Infer para o exemplo mínimo da Listagem B.17.

```
Found 1 issue

memory_leak_true_positive_realloc_infer.c:14: error:
MEMORY_LEAK memory dynamically allocated by call to
`malloc()` at line 12, column 15 is not reachable
after line 14, column 2.
  12.       st *new = malloc(sizeof(st));
  13.     if(!new) return -1;
  14. >   new = realloc(new,2*sizeof(st));
  15.     free(new);
  16.     return 0;

Summary of the reports

  MEMORY_LEAK: 1
```

Listagem B.20: Exemplo mínimo de um erro do tipo fuga de memória utilizando uma função `realloc` corrigido.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct st{
   int value;
}st;

int main(){
  st *tmp;
    st *new = malloc(sizeof(st));
  if(!new) return -1;
  tmp = realloc(new,2*sizeof(st));
  if(tmp)
  {
    new = tmp;
  }
  free(new);
  return 0;
}
```

B.3 Desreferência nula

O Infer é a única ferramenta que reporta um erro quando não é feita a verificação da chamada a uma função de alocação de memória. Portanto, no exemplo mínimo da Listagem B.21, apenas foram obtidos relatórios de resultados do Infer (Listagem B.22). Na Listagem B.23 o erro é corrigido e não deixa de ser reportado pelo Infer.

Listagem B.21: Exemplo mínimo de um erro do tipo desreferência nula.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct st{
   int value;
}st;

st * create(int value){
   st *new = malloc(sizeof(st));
   new -> value = value;
   return new;
}

int main(){
    st *new = create(5);
    free(new);
    return 0;
}
```

Listagem B.22: Relatório de erros devolvido pela ferramenta Infer para o exemplo mínimo da Listagem B.21.

```
Found 1 issue

null_dereference_true_positive_mallocverification.c:13:
error: NULL_DEREFERENCE pointer `new` last assigned on
line 12 could be null and is dereferenced at line 13,
column 4.
  11.   st * create(int value){
  12.       st *new = malloc(sizeof(st));
  13. >     new -> value = value;
  14.       return new;
  15.   }

Summary of the reports

  NULL_DEREFERENCE: 1
```

Listagem B.23: Exemplo mínimo de um erro do tipo desreferência nula corrigido.

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct st{
   int value;
}st;

st * create(int value){
   st *new = malloc(sizeof(st));
   if(!new) return NULL;
   new -> value = value;
   return new;
}

int main(){
    st *new = create(5);
    free(new);
    return 0;
}
```

O CSA e o Infer não são capazes de reconhecer quando é feita uma verificação através de asserções. Na Listagem B.24 está representado um exemplo da utilização de uma asserção para verificar o valor de um apontador que é, mais tarde, desreferenciado. Na Listagem e na Listagem estão representados os relatórios de resultados das CSA e Infer, respetivamente.

Listagem B.24: Exemplo mínimo de um erro do tipo desreferência nula.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct st{
   int value;
}st;

st * create(int value){
   st *new = malloc(sizeof(st));
   assert(!new);
   new -> value = value;
   return new;
}

int main(){
    st *new = create(5);
    free(new);
    return 0;
}
```

Listagem B.25: Relatório de erros devolvido pela ferramenta Infer para o exemplo mínimo da Listagem B.24.

```
Found 1 issue

asserts_null_dereference.c:13: error: NULL_DEREFERENCE
  pointer `new` last assigned on line 11 could be null and is
  dereferenced at line 13, column 4.
  11.       st *new = malloc(sizeof(st));
  12.       assert(!new);
  13. >     new -> value = value;
  14.       return new;
  15.   }

Summary of the reports

  NULL_DEREFERENCE: 1
```

Listagem B.26: Relatório de erros devolvido pela ferramenta CSA para o exemplo mínimo da Listagem B.24.

```
asserts_null_dereference.c:13:17: warning: Access to field 'value'
results in a dereference of a null pointer (loaded from variable 'new')
   new -> value = value;
   ~~~           ^
1 warning generated.
```