

# Uma análise comparativa de ferramentas de análise estática para deteção de erros de memória

Patrícia Monteiro, João Lourenço, e António Ravara

Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa (FCT-NOVA)  
NOVA Laboratory for Computer Science and Informatics (NOVA LINCS)  
ps.monteiro@campus.fct.unl.pt  
{joao.lourenco, aravara}@fct.unl.pt

**Resumo** As falhas de software estão com frequência associadas a acidentes com graves consequências económicas e/ou humanas, pelo que se torna imperioso investir na validação do software, nomeadamente daquele que é crítico. Este artigo 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 reportados.

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

## 1 Introdução

*“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)

A crescente necessidade de desenvolvimento de software cada vez mais complexo, no menor tempo possível e com baixo custo, conduz ao aumento da densidade de erros, sendo o controlo de qualidade frequentemente assegurada com base em testes e revisão manual de código. Estas técnicas, apesar de eficazes, não permitem garantir a ausência total de erros num programa.

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. As falhas de software crítico são muitas vezes associadas a desastres com graves consequências económicas e/ou humanas. São exemplos disso casos como a autodestruição do Ariane 5 (1996) [\[1\]](https://around.com/ariane.html) e do Mars Climate Orbiter

<sup>1</sup> <https://around.com/ariane.html>

(1999) [S<sup>+</sup>99], o bloqueio do terminal 5 no Aeroporto de Londres-Heathrow (2008)<sup>2</sup> e, mais recentemente, o erro encontrado pela Amazon no Jedis (2018)<sup>3</sup>

No software escrito em C/C++ é comum a existência de erros de memória, tais como desreferências inválidas, acesso a variáveis não inicializadas e fugas e operações inválidas de libertação de memória. Este tipo de erros são de difícil deteção pois normalmente conduzem ao comportamento indefinido do programa [Reg10], isto é, tanto podem causar a sua falha imediata como permitir que este continue a funcionar de forma silenciosamente defeituosa.

A verificação e validação de software é feita, essencialmente, através de três técnicas: revisão manual de código [Gee16], análise automática (dinâmica [Bal99] ou estática [W<sup>+</sup>95]) e semi-automática com provadores de teoremas [Duf91]. As duas últimas são rigorosas e permitem analisar a totalidade do código, verificando propriedades que são definidas matematicamente e identificando todas as situações em que as mesmas são violadas: *os erros*.

A análise automática de software exige menos esforço por parte das equipas de desenvolvimento, fator importante no contexto atual da indústria. A análise dinâmica implica a execução do programa ou de uma sua representação, enquanto que a estática analisa o programa sem executar o código fonte ou uma sua representação. A utilização precoce de técnicas de análise estática permite a identificação de erros numa fase inicial do desenvolvimento, o que reduz consideravelmente os custos [Pat06]. Como estas ferramentas utilizam frequentemente um processo de sobre-aproximação das propriedades que pretendem verificar, reportam com frequência *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.

Interessa-nos estudar a viabilidade de usar ferramentas automáticas para procurar erros de memória, e o objetivo deste artigo é apresentar uma análise comparativa preliminar de quatro ferramentas de análise estática de código aberto: Cppcheck<sup>4</sup> [Mar17], Clang Static Analyzer<sup>5</sup>, Infer<sup>6</sup> [CD11] e Predator<sup>7</sup> [DPV13]. O processo e critérios de seleção destas ferramentas encontra-se descrito na Secção 2.1. Através desta análise pretende-se identificar o tipo de padrões de erros de memória que as ferramentas conseguem detetar, identificando aquelas que são funcionalmente equivalentes, por reportarem os mesmos erros, e as que são complementares, por reportarem erros distintos.

Neste artigo apresenta-se a análise de dois programas, um de pequena e outro de média dimensão, estando a decorrer a análise de outros dois programas de grande dimensão. Como seria de esperar, verificou-se que havia considera-

<sup>2</sup> <http://www.computerweekly.com/news/2240086013/>

[British-Airways-reveals-what-went-wrong-with-Terminal-5](http://www.computerweekly.com/news/2240086013/British-Airways-reveals-what-went-wrong-with-Terminal-5)

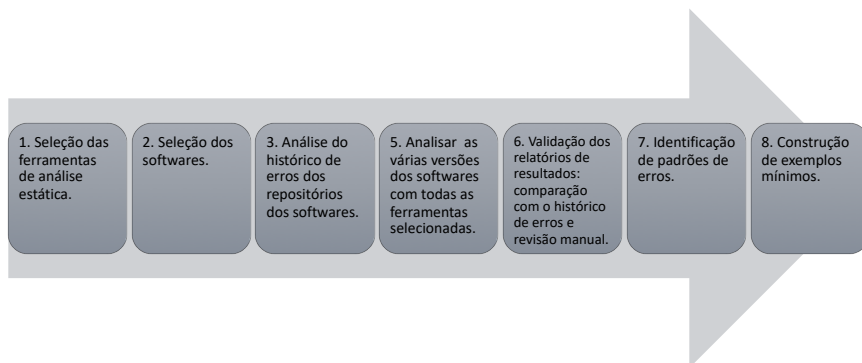
<sup>3</sup> <https://github.com/xetorthio/jedis/issues/1747>

<sup>4</sup> <https://github.com/danmar/cppcheck>

<sup>5</sup> <http://clang.llvm.org/>

<sup>6</sup> <https://github.com/facebook/infer>

<sup>7</sup> <https://github.com/kdudka/predator>



**Figura 1.** Etapas seguidas durante a realização da experiência.

velmente mais erros de memória reportados no software de média dimensão que no de pequena dimensão. Naturalmente, o número de falsos positivos também aumentou. No entanto, através dos resultados obtidos foi possível construir exemplos mínimos que permitiram perceber de forma clara o tipo de padrões de erros identificados pelas várias ferramentas. Os padrões de erros descobertos permitiram identificar as diferenças e falhas na análise das ferramentas. Além disso, foram descobertos novos erros nos programas escrutinados, sendo que os mesmos foram reportados nos respectivos repositórios.

## 2 Metodologia

A Figura 1 apresenta o processo seguido durante este trabalho. O processo teve início com a identificação e seleção das ferramentas de análise estática a estudar (Secção 2.1) e dos programas alvo (Secção 2.2). Concluída a fase de seleção das ferramentas e dos programas, seguiu-se a análise dos mesmos. Inicialmente, foram recolhidas informações sobre o número e tipo de erros de memória já identificados nos programas, pesquisando os repositórios dos mesmos. Os erros reportados como questões de utilizadores (i.e., *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; e 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. Estes foram, posteriormente, verificados através da revisão manual do código. Por fim, com as informações recolhidas foi

**Tabela 1.** Funcionalidades das ferramentas de análise estática.

Ferramenta	Operações inválidas			
	Fuga de memória	Desreferências inválidas	de libertação de memória	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

possível identificar padrões de erros reconhecidos pelas ferramentas e construir exemplos mínimos para cada um deles.

## 2.1 Escolha de ferramentas

A escolha das ferramentas de análise estática a utilizar nesta experiência 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; e iv) identificarem pelo menos dois dos seguintes erros: desreferenciação inválida, operação de libertação inválida, fuga de memória e variáveis não inicializadas.

Assim, numa primeira seleção foram identificadas 32 ferramentas, que foram dispostas na Tabela 5 (ver Apêndice A), onde cerca de metade corresponde a ferramentas de código aberto e outra metade a ferramentas comerciais. Ficámos com uma seleção de apenas 10 ferramentas que eram simultaneamente projetos ativos e de código aberto. Posteriormente, estas 10 ferramentas foram classificadas relativamente às suas funcionalidades de verificação de software (Tabela 1), tendo sido selecionadas as ferramentas que garantissem pelo menos duas das funcionalidades listadas.

Foram identificadas 6 ferramentas que cumpriam com todos os critérios referidos (cerca de 19% da lista de ferramentas inicial), que foram dispostas na Tabela 2 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 no Apêndice B. Além disso, na Tabela 2 é 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) excepto o Predator, que apenas funciona em Linux. Relativamente à facilidade de utilização, metade das ferramentas selecionadas não precisa de uma função

**Tabela 2.** Características das ferramentas de análise estática.

Ferramenta	Teoria	Correção	Instalação		Utilização			
			Fornece executável	Multi-plataforma	Requere main	Requere 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

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.

No final do processo de seleção obtiveram-se as seguintes ferramentas: Cppcheck [Mar17], Clang Static Analyzer<sup>8</sup>, Frama-C [C+17], Infer [CD11], Predator [DPV13] e UNO [Hol02]. 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.

## 2.2 Escolha de projetos a analisar

Através de uma pesquisa de software aberto no GitHub, seleccionámos 16 projetos escritos nas linguagens C/C++, que foram 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.

**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). 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*.

<sup>8</sup> <http://clang.llvm.org/>

**Tabela 3.** Classificação das dimensões dos Programas.

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

**Falha:** impacto dos erros identificados no software, 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 software. 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 depois ordenados por prioridade. Os com a mesma dimensão e prioridade foram ordenados por popularidade. Na Tabela 4 estão listados, de forma já ordenada, todos os programas analisados e respetivas características. Os quatro 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, tendo eles prioridade igual a 1 e elevada popularidade:

**Simple Dynamic String (SDS)**<sup>9</sup> biblioteca de *strings* dinâmicas, projetada para aumentar as funcionalidades limitadas das *strings* da biblioteca C;

**Beanstalkd**<sup>10</sup> gestor de tarefas para aplicações distribuídas;

**Tmux**<sup>11</sup> multiplexador de terminal, que permite que uma série de terminais possam ser acedidos e controlados a partir de um único terminal;

**Memcached**<sup>12</sup> sistema distribuído de cache em memória, que é frequentemente utilizado para acelerar sites dinâmicos, colocando os dados dos mesmos em cache para reduzir o número de vezes que uma fonte de dados externa precisa de ser acedida.

### 3 Experimentação e análise

#### 3.1 Relatório de resultados

Neste artigo apresentamos os resultados referentes a dois programas. O SDS é um software de pequena dimensão com apenas 2 versões no seu repositório.

<sup>9</sup> <https://github.com/antirez/sds>

<sup>10</sup> <https://github.com/kr/beanstalkd>

<sup>11</sup> <https://github.com/tmux/tmux>

<sup>12</sup> <https://github.com/memcached/memcached>

**Tabela 4.** Características dos programas.

Software	Prioridade	Dimensão	Popularidade (# Estrelas)	Versões	Erros de	Erros	Casos ( <i>issues</i> )	Manipulação	
					memória	causam	sobre erros	# Operações	# Apontadores
					(%) total erros	(%) erros mem.	(%) total casos		
Simple Dynamic String	1	Pequena	2215	2	0	0	23	121	71
TinyVM	1	Pequena	1254	1	25	4	25	64	115
Twencache	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	20	8	11	3	1116	918
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

Os respetivos *commits* não reportam erros de memória, mas existem vários erros deste tipo identificados por utilizadores (i.e., *issue*), 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. O Beanstalkd é um software 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.

Os erros identificados no SDS e no Beanstalkd, quer pelos programadores e utilizadores, quer pelas ferramentas, encontram-se listados nas Tabelas 9 e 10 (Apêndice C). No total foram analisados 30 erros no SDS, dos quais 24 foram identificados pelas ferramentas, e 188 erros no Beanstalkd, dos quais 175 foram identificados pelas ferramentas. De notar que, na Tabela 10 foram omitidos os 135 falsos positivos identificados pelo Predador. Na secção seguinte é feita uma descrição da análise dos relatórios obtidos.

### 3.2 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 foi 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.

**Simple Dynamic String (SDS).** Na versão 1 foram reportados 7 erros no repositório, sendo um desses um falso positivo identificado por um utilizador. Na versão 2 foram reportados 2 erros, mas um já existia na versão 1. Por outro lado, as ferramentas encontram um total de 24 erros diferentes, sendo que 21 desses não foram reportados. No entanto, apenas 6 dos 21 erros poderão ser classificados como reais. Assim, no SDS foram identificados 26 erros, dos quais 13 são reais. Os erros reais que encontramos na última versão e que reportámos são:

- *sds.c:1123: error: null dereference*, identificado pelo Infer<sup>13</sup>
- *sds.c:1240: error: dead store*, identificado pelo Clang e pelo Infer<sup>14</sup>

Na Figura 2(a) está representado um gráfico com as percentagens dos vários tipos de erros identificados pelas ferramentas no SDS. 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 Clang Static Analyzer e Predator como erros, que nós classificámos como falsos positivos.

Verificámos que o Clang Static Analyzer apresentou uma percentagem relativamente baixa de falsos positivos, mas verificou-se também 30% de falsos negativos. Esta percentagem deve-se ao facto da Clang não identificar um padrão de erros relevante, nomeadamente a verificação dos resultados de operações de alocação de memória (i.e., `malloc`, `calloc` e `realloc`).

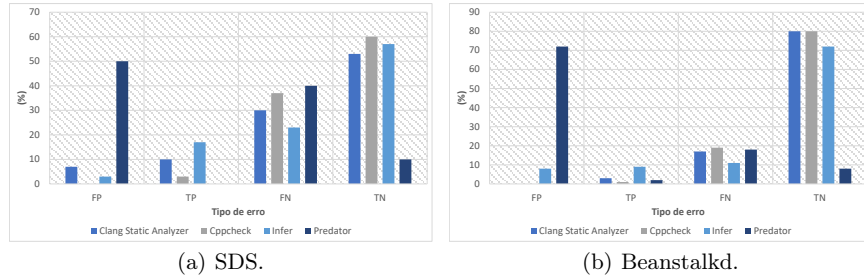
Por princípio, a ferramenta Predator não permite chamadas de funções externas, a fim de excluir qualquer efeito colateral que possa potencialmente quebrar a segurança da memória. As únicas funções externas permitidas são aquelas que o Predator reconhece como funções integradas e as modela apropriadamente, provando a segurança da memória (`malloc`, `free`, e algumas funções da biblioteca do C tais como `memset`, `memcpy` e `memmove` [DPV13]). 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.

Apesar de falhar na identificação de alguns erros, o Infer é a ferramenta que devolve uma maior percentagem de verdadeiros positivos e a menor percentagem de falsos negativos neste software. Como a análise realizada pelo Infer consiste numa execução simbólica do código, mantendo uma *heap* simbólica, 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, porque a tentativa de prova não faz sentido. Outra razão para

<sup>13</sup> <https://github.com/antirez/sds/issues/99>

<sup>14</sup> <https://github.com/antirez/sds/issues/100>





**Figura 2.** Percentagem de cada tipo de erros identificado nos programas.

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 da análise chegar ao fim [CD11], 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 reportado, mas nesse processo acabar por eliminar erros reais gerando falsos negativos.

**Beanstalkd.** Este software tem uma dimensão e complexidade maior que o SDS. Como o Predator não está preparado para analisar programas muito complexos [DPV13], a análise do Beanstalkd revelou-se pouco conclusiva. 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 135 falsos positivos (cerca de 72% 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 e verdadeiros negativos. Estes resultados devem-se ao facto desta ferramenta fazer filtragem de falsos positivos. Por fim, o Clang Static Analyzer 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.

Os novos erros identificados e classificados como verdadeiros positivos no repositório do Beanstalkd foram os seguintes:

- *prot.c:501: error: null dereference*, identificado pelo Infer<sup>15</sup>
- *testheap.c:222: error: memory leak*, identificado pelo Cppcheck e Predator<sup>16</sup>

<sup>15</sup> <https://github.com/kr/beanstalkd/issues/384>

<sup>16</sup> <https://github.com/kr/beanstalkd/issues/382>

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

## 4 Conclusões

Como se pode verificar nas Tabelas 9 e 10 do Apêndice F é útil usar as 4 ferramentas, pois obtém-se resultados complementares: cada ferramenta identificou erros que nenhuma das outras identificou. Este facto valida a seleção feita.

É também importante salientar que a utilização destas ferramentas não pesa significativamente no desenvolvimento de software, pois os tempos que cada uma leva a analisar os softwares escolhidos são muito reduzidos, como se pode ver nas Tabelas 11 e 12 do Apêndice G: vão de alguns segundos, no caso do SDS, a no máximo pouco mais de um minuto, no caso do Beanstalkd.

Note-se que vários dos erros encontrados pelas ferramentas estiveram presentes nos softwares por longos períodos de tempo, como se vê nas Tabelas 13 e 14 do Apêndice H: o SDS teve erros que foram corrigidos só passados dois anos e tem erros que presentes há cerca de quatro anos; o Beanstalkd teve erros que foram corrigidos só passados quatro anos e tem erros presentes há quase dez anos.

A deteção destes erros pelas ferramentas foi no entanto muito rápida (como se referiu acima) e a verificação de que se tratavam de erros reais levou poucas horas. É então muito vantajosa a utilização destas ferramentas de análise estática no processo de desenvolvimento de software para evitar erros de memória.

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 8 (Apêndice D) estão representados os padrões identificados para cada tipo de erro. Os exemplos mínimos e respetivos relatórios de resultados podem ser consultados no Apêndice E.

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 Clang Static Analyzer não considera esta situação 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))`. Na versão mais recente do Infer (versão 0.14.0, lançada no dia 1 de Maio de 2018) este defeito já foi corrigido. No Apêndice E deste artigo encontra-se um exemplo mínimo da situação descrita.

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

tados pelo Cppcheck e Infer são diferentes. O Cppcheck identifica uma possível fuga de memória, enquanto que o Infer identifica uma desreferência nula. Isto acontece porque 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.

Este tipo de ferramentas está já a ser incluído no processo de desenvolvimento de alguns programas de grande dimensão e relevância, como o LibreOffice<sup>17</sup>. Atualmente, este software usa duas ferramentas, o Cppcheck e o Coverity [Lla17]. Segundo o relatório disponibilizado, 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% [Proa]. No caso do Cppcheck, segundo o último relatório disponível (de 27 de Janeiro de 2018), foram identificados através desta ferramenta cerca de 6 mil erros no repositório do LibreOffice [Prob].

O trabalho futuro passa pela análise dos programas de grande dimensão já selecionados (Tmux e Memcached). Os dados recolhidos dessa análise serão depois utilizados para identificar novos padrões de erros e construir exemplos mínimos, tal como foi feito para o SDS e para o Beanstalkd. Desta forma espera-se obter novos dados para o desenvolvimento do estudo comparativo das ferramentas.

**Agradecimentos.** *Este trabalho foi suportado pela FCT-NOVA e parcialmente financiado pela FCT-MCTES e pelo programa POCI-COMPETE2020 nos projetos UID/CEC/04516/2013 e PTDC/CCI-COM/32456/2017.*

## Referências

- All70. Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7), 1970.
- Bal99. Thoms Ball. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6), 1999.
- BCGG13. R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb. Symbolic path-oriented test data generation for floating-point programs. In *ICST*. IEEE, 2013.
- BCLR04. Tom Ball, Byron Cook, Vladimir Levin, and Sriram Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *iFM*, volume 2999 of *LNCS*, pages 1–20. 2004.
- Bol16. C. Bolduc. Lessons learned: Using a static analysis tool within a continuous integration system. In *ISSREW*. IEEE, 2016.
- C<sup>+</sup>17. Loïc Correnson et al. *Frama-C User Manual*, 2017. <https://frama-c.com/download.html>
- CC77. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*. ACM, 1977.
- CD11. Cristiano Calcagno and Dino Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NFM*, volume 6617 of *LNCS*. Springer, 2011.

<sup>17</sup> <https://github.com/LibreOffice/core>

- DPV13. Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Byte-Precise Verification of Low-Level List Manipulation. In *SAS*, volume 7935 of *LNCS*. Springer, 2013.
- DS07. David Delmas and Jean Souyris. Astrée: From Research to Industry. In *SAS*, volume 4634 of *LNCS*. Springer, 2007.
- Duf91. David A. Duffy. *Principles of Automated Theorem Proving*. John Wiley & Sons, 1991.
- EGHT94. David Evans, John Guttag, James Horning, and Yang Meng Tan. Lclint: A tool for using specifications to check code. In *FSE*. ACM, 1994.
- EL02. D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1), 2002.
- F<sup>+</sup>07. Ansgar Fehnker et al. Goanna—A Static Model Checker. In *FMICS/PDMC*, volume 4346 of *LNCS*. Springer, 2007.
- G<sup>+</sup>08. Eric Goubault et al. Static Analysis of the Accuracy in Control Systems: Principles and Experiments. In *FMICS*, volume 4916 of *LNCS*. Springer, 2008.
- Gee16. Trisha Gee. *What to Look for in a CodeReview*. JetBrains Technical Series, 2016.
- Gim14. J. Gimpel. Software that checks software: The impact of pc-lint. *IEEE Software*, 31(1), 2014.
- Hen78. M. A. Hennell. An experimental testbed for numerical software. *The Computer Journal*, 21(4), 1978.
- Hoa69. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10), 1969.
- Hol02. Gerard J. Holzmann. Static source code checking for user-defined properties. In *Integrated Design and Process Technology*, volume 2, 2002.
- Hol17. Gerard J. Holzmann. Cobra: Fast structural code checking. In *SPIN*. ACM, 2017.
- Lla17. Mel Llaguno. 2017 coverity scan report. Technical report, Synopsys, 2017. <https://www.synopsys.com/blogs/software-security/2017-coverity-scan-report-open-source-security/>.
- Mar17. Daniel Marjamaki. *Cppcheck 1.81*, 2017. <http://cppcheck.sourceforge.net/>.
- NFH05. Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes*, 30(4), 2005.
- Pat06. Ron Patton. *Software testing*. Pearson Education India, 2006.
- Proa. Coverity Project. Coverity Scan – Static Analysis. <https://scan.coverity.com/projects/211>
- Prob. LibreOffice Project. Cppcheck - HTML report – LibreOffice 2018-01-27. [https://dev-builds.libreoffice.org/cppcheck\\_reports/master/index.html](https://dev-builds.libreoffice.org/cppcheck_reports/master/index.html)
- PRQ17. PRQA Programming Research. QA-C static analyzer. Technical report, 2017. <http://www.prqa.com/static-analysis-software/qac-qacpp-static-analyzers/>.
- Reg10. John Regehr. A Guide to Undefined Behavior in C and C++, Part 1 – Embedded in Academia. <https://blog.regehr.org/archives/213>, 2010.
- Rey02. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*. IEEE, 2002.
- RVP06. Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In *Ada-Europe 2006*, volume 4006 of *LNCS*. Springer, 2006.

- S<sup>+</sup>99. Arthur G. Stephenson et al. Mars Climate Orbiter Mishap Investigation Board Phase I Report November 10, 1999. Technical report, NASA, 1999.
- SMM12. Pavel Shved, Mikhail Mandrykin, and Vadim Mutilin. Predicate Analysis with BLAST 2.7. In *TACAS*, volume 7214 of *LNCS*. Springer, 2012.
- W<sup>+</sup>95. B. A. Wichmann et al. Industrial perspective on static analysis. *Software Engineering Journal*, 10(2), 1995.
- Whe17. David A. Wheeler. Flawfinder. Technical report, 2017. <https://www.dwheeler.com/flawfinder/#documentation>
- Wis07. Klaus Wissing. Static Analysis of Dynamic Properties-Automatic Program Verification to Prove the Absence of Dynamic Runtime Errors. In *INFORMATIK*, volume 110 of *LNI*. GI, 2007.
- XA05. Yichen Xie and Alex Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *CAV*, volume 3576 of *LNCS*. Springer, 2005.

## A Escolha das ferramentas de análise estática

Tabela 5. Ferramentas de análise estática.

Ferramenta	Código aberto	Ativo
AdLint <sup>18</sup>	✓	x
Astrée [DS07]	x	✓
Axivion Bauhaus Suite [RVP06]	x	✓
BLAST [SMM12]	✓	x
Cppcheck [Mar17]	✓	✓
Cppdepend <sup>19</sup>	x	✓
Cpplint <sup>20</sup>	✓	✓
Clang Static Analyzer <sup>21</sup>	✓	✓
Cobra [Hol17]	✓	✓
Codacy <sup>22</sup>	x	✓
CodeSonar <sup>23</sup>	x	✓
Coverity [Lla17]	x	✓
ECLAIR [BCGG13]	x	✓
Flawfinder [Whe17]	✓	✓
Fluctuat [G <sup>+</sup> 08]	x	✓
Frama-C [C <sup>+</sup> 17]	✓	✓
Goanna [F <sup>+</sup> 07]	x	✓
Infer [CD11]	✓	✓
Klocwork Static Code Analysis [Bol16]	x	✓
LCLint [EGHT94]	✓	x
LDRA Testbed [Hen78]	x	✓
Parasoft C/C++ test <sup>24</sup>	x	✓
PC-Lint [Gim14]	x	✓
Polyspace [Wis07]	x	✓
Predator [DPV13]	✓	✓
PVS-Studio <sup>25</sup>	x	✓
PRQA QAC [PRQ17]	x	✓
Saturno [XA05]	✓	x
SLAM project [BCLR04]	x	x
Splint [EL02]	✓	x
Uno [Hol02]	✓	✓
VisualCodeGrepper <sup>26</sup>	✓	✓

<sup>18</sup> <http://adlint.sourceforge.net/><sup>19</sup> <https://www.cppdepend.com/><sup>20</sup> <https://github.com/google/styleguide/tree/gh-pages/cpplint><sup>21</sup> <http://clang.llvm.org/><sup>22</sup> <https://www.codacy.com/>

## B Teorias

### B.1 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 [CC77].

### B.2 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 [NFH05].

### B.3 Grafo de Controle de Fluxo (CFG):

Consiste num grafo orientado, no qual são representados todos os caminhos lógicos que podem ser percorridos durante a execução de um programa [All70].

### B.4 Lógica de Separação (SL):

Uma lógica matemática, que consiste numa extensão da lógica de Hoare [Hoa69] 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 [Rey02].

### B.5 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 [DPV13].

<sup>23</sup> <https://www.grammatech.com/products/codesonar>

<sup>24</sup> <https://www.parasoft.com/products/ctest>

<sup>25</sup> <https://www.viva64.com/en/pvs-studio/>

<sup>26</sup> <https://github.com/nccgroup/VCG>





## C Relatórios de erros

**Tabela 6.** Resultados da análise do software Beanstalkd.

Erro	Versão		Reportado	Falso positivo	Ferramenta
	Identificado	Corrigido			
beanstalkd.c:667: error: memory leak	0.3	0.4	✓	x	x
net.c:28: error: dead store	0.3	1.4	x	x	Clang Static An. Infer
beanstalkd.c:41: error: resource leak	0.3	1.5	x	x	Infer
reserve.c:51: error: null dereference	0.3	1.5	x	x	Infer
prot.c:140: error: null dereference	0.3	1.5	x	x	Infer
beanstalkd.c:395: error: null dereference	0.3	0.5	x	x	Infer
beanstalkd.c:737: error: memory leak	0.3	0.5	x	x	Infer
beanstalkd.c:205: error: memory leak	0.4	0.5	✓	x	x
tube.c:60: error: memory leak	0.8	1.2	✓	x	x
ms.c:56: error: memory leak	0.8	x	x	✓	Infer
prot.c:320: error: null dereference	0.8	1.5	x	x	Infer
prot.c:374: error: null dereference	0.8	1.5	x	x	Infer
prot.c:946: error: dead store	0.8	x	x	✓	Infer
prot.c:1134: error: dead store	0.8	x	x	✓	Infer
prot.c:1170: error: dead store	0.8	x	x	✓	Infer
prot.c:1230: error: dead store	0.8	x	x	✓	Infer
prot.c:1388: error: dangling pointer	1.2	1.3	✓	x	x
job.c:170: error: memory leak	1.2	1.4	✓	x	x
prot.c:1493: error: memory leak	1.2	1.4	✓	x	x
job.c:48: error: dead store	1.2	x	x	✓	Infer
tube.c:86: error: dead store	1.2	x	x	✓	Infer
job.c:95: error: dead store	1.2	x	x	✓	Infer
conn.c:224: error: dead store	1.2	x	x	✓	Infer
prot.c:1583: error: null dereference	1.2	x	x	x	Infer
cut.c:222: error: resource leak	1.2	1.5	x	x	Cppcheck
job:52:error: invalid dereference	1.2	x	x	✓	Predator
binlog.c:176: error: buffer overflow	1.4.1	1.4.2	✓	x	x
binlog.c:215: null dereference	1.4.1	1.4.2	x	x	Clang Static An.
binlog.c:723: error: null dereference	1.4.1	1.5	x	x	Infer
job.c:70: error: memory leak	1.4.2	1.4.3	✓	x	Predator
prot.c:974: error: memory leak	1.4.2	1.4.3	✓	x	x
prot.c:1017: error: memory leak	1.4.3	1.4.4	✓	x	x
net.c:31: error: dead store	1.5	1.9	x	x	Clang Static An. Infer
heap.c:98: error: memory leak	1.5	x	x	✓	Infer
walg.c:293: error: memory leak	1.5	x	x	✓	Infer
walg.c:293: error: resource leak	1.5	x	x	✓	Infer
walg.c:357: error: memory leak	1.5	x	x	✓	Infer
walg.c:357: error: resource leak	1.5	x	x	✓	Infer
walg.c:426: error: resource leak	1.5	x	x	✓	Infer
prot.c:514: error: null dereference	1.5	1.8	x	x	Infer
prot.c:554: error: null dereference	1.5	1.8	x	x	Infer
file.c:204: error: null dereference	1.6	1.7	✓	x	Clang Static An.
file.c:325: error: null dereference	1.6	1.7	✓	x	Clang Static An.
prot.c:501: error: null dereference	1.6	x	x	x	Infer
conn.c:244: error: invalid free	1.8	1.9	✓	x	x
integ-test.c:343: error: invalid free	1.8	1.9	✓	x	x
prot.c:710: error: invalid free	1.8	1.9	✓	x	x
darwin.c:84: error: dead store	1.8	x	x	x	Infer
walg.c:416: dead store	1.9	1.10	✓	x	Clang Static An. Infer
testheap.c:222: error: memory leak	1.10	x	x	x	Cppcheck Predator

**Tabela 7.** Resultados da análise do software SDS.

Erro	Versão		Reportado	Falso positivo	Ferramenta
	Identificado	Corrigido			
sds.c:303: error: memory leak	1	2	✓	x	x
sds.c:470: error: memory leak	1	x	✓	✓	x
sds.c:882: error: memory leak	1	2	✓	x	x
sds.c:881: error: memory leak	1	2	✓	x	x
sds.c:159: error: memory leak	1	2	✓	x	Cppcheck
sds.c:160: error: null dereference	1	2	✓	x	Infer
sds.c:717: error: null dereference	1	x	✓	x	x
sds.c:817: error: null dereference	1	x	x	x	Infer
sds.c:891: null pointer	1	2	x	x	Clang Static An.
sds.c: 894: null pointer	1	2	x	x	Clang Static An.
sds.c:63: error: invalid dereference	1	x	x	✓	Predator
sds.c:147: error: invalid dereference	1	x	x	✓	Predator
sds.c:793: error: memory leak	1	x	x	✓	Predator
sds.c:92: error: null dereference	2	x	✓	x	Infer
sds.c:1123: error: null dereference	2	x	x	x	Infer
sds.c:580: error: dead store	2	x	x	✓	Infer
sds.c:136: error: invalid dereference	2	x	x	✓	Predator
sds.c:98: error: dereferencing object of size 8B out of bounds	2	x	x	✓	Predator
sds.c:96: error: dereferencing object of size 4B out of bounds	2	x	x	✓	Predator
sds.c:94: error: dereferencing object of size 2B out of bounds	2	x	x	✓	Predator
sds.c:92: error: dereferencing object of size 1B out of bounds	2	x	x	✓	Predator
sds.c:161: error: invalid free	2	x	x	✓	Clang Static An. Predator
sds.c:161: error: memory leak	2	x	x	✓	Predator
sds.c:1121: error: memory leak	2	x	x	✓	Predator
sds.h:87: error: dereference NULL value with offset -1B	2	x	x	✓	Predator
sds.h:197: error: invalid dereference	2	x	x	✓	Predator
sds.c:229: invalid free	2	x	x	✓	Clang Static An.
sds.c:1240: dead store	2	x	x	x	Clang Static An. Infer



## D Padrões de erros

Tabela 8. Padrões de erros.

Padrão	Erro	Ferramenta			
		Cppcheck	Clang Static Analyzer	Infer	Predator
Apontador inicializado com o valor NULL e desreferenciado na mesma função.	Desreferência nula	✓	✓	✓	✓
Apontador inicializado com o valor NULL noutra função e desreferenciado.	Desreferência nula	x	✓	✓	✓
Falta de verificação após uma alocação de memória.	Desreferência nula	x	x	✓	x
O valor passado como argumento à função de alocação de memória ( <code>malloc</code> ou <code>calloc</code> ) é do tipo <code>*ptr</code> .	Desreferência nula Fuga de memória Operação de libertação inválida	x	✓	x	✓
O valor escrito no endereço é inicializado com o valor NULL ou 0 e nunca é usado.	Valor do endereço nunca é usado	x	x	✓	x
O valor escrito no endereço é inicializado com um valor diferente de NULL e 0 e nunca é usado.	Valor do endereço nunca é usado	x	✓	✓	x
O valor do apontador é alterado através da função <code>realloc</code> , sem garantir que esta tem sucesso (sem chamadas a funções).	Fuga de memória	✓	x	✓	x
O valor do apontador é alterado através da função <code>realloc</code> , sem garantir que esta tem sucesso (com chamadas a funções).	Fuga de memória	x	x	✓	x
Alocação e libertação de memória explícita.	Fuga de memória	✓	✓	✓	✓
Alocação e libertação de memória não explícita.	Fuga de memória	x	✓	✓	✓
Libertação da estrutura antes da libertação da memória alocada para apontadores que fazem parte da mesma.	Fuga de memória	x	✓	✓	✓

## E 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.

### E.1 Dead store

A Listagem 1.1 corresponde ao exemplo mínimo de um falso positivo identificado pelo Infer (Listagem 1.2). 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 1.3 o erro é corrigido e o Infer não devolve qualquer tipo de resultado. Por fim, na Listagem 1.4 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 1.5 e Figura 3).

**Listing 1.1.** Exemplo mínimo de um erro *dead store* classificado como falso positivo.

```
#include <stdio.h>

int main () {
    int var = 20;
    int *ptr_a = NULL;

    ptr_a = &var;
    printf("Value:%d\n",*ptr_a);

    return 0;
}
```

**Listing 1.2.** Relatório de resultados do Infer para o exemplo mínimo da Listagem 1.1

```
Found 1 issue

dead_store_false_positive_infer.c:7: error: DEAD_STORE
The value written to &ptr_a is never used.
5.   int main () {
6.     int var = 20;
7. >  int *ptr_a = NULL;
8.
9.     ptr_a = &var;

Summary of the reports

DEAD_STORE: 1
```

**Listing 1.3.** Exemplo mínimo de um erro do tipo *dead store* corrigido.

```
#include <stdio.h>

int main () {
    int var = 20;
    int *ptr_a = &var;
    printf("Value:%d\n",*ptr_a);

    return 0;
}
```

**Listing 1.4.** Exemplo mínimo de um erro do tipo *dead store* classificado como verdadeiro positivo.

```
#include <stdio.h>

int main () {
    int a = 20;
    int b = 10;
    int *ptr_a = &a;

    ptr_a = &b;
    printf("Value:%d\n",*ptr_a);

    return 0;
}
```

**Listing 1.5.** Relatório de resultados da ferramenta Infer para o exemplo mínimo da Listagem [1.4](#)

```
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 3.** Relatório de erros devolvido pela ferramenta Clang Static Analyzer para o exemplo mínimo da Listagem 1.4

```

1  /*Exemplo minimo de um suposto erro do tipo DEAD STORE
2  identificado pela ferramenta Infer - falso positivo*/
3  #include <stdio.h>
4
5  int main () {
6      int a = 20;
7      int b = 10;
8      int *ptr_a = &a;
9
10     ptr_a = &b;
11     printf("Value:%d\n", *ptr_a);
12
13     return 0;
14 }

```

Value stored to 'ptr\_a' during its initialization is never read

## E.2 Fuga de memória

**Argumento passado ao malloc.** Na Listagem 1.6 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 1.7 e da Figura 4. Na Listagem 1.8 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 1.9 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 1.10). Além disso, o Predator e o Clang Static Analyzer (Listagem 1.11 e Figura 5) também conseguem identificar o erro, tal como já acontecia no exemplo mínimo anterior.

**Listing 1.6.** 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;
}

```

**Listing 1.7.** Relatório de resultados da ferramenta Predator para o exemplo mínimo da Listagem [1.6](#)

```

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]

```



**Figura 4.** Relatório de erros devolvido pela ferramenta Clang Static Analyzer para o exemplo mínimo da Listagem 1.6

```

1  /*Exemplo mínimo de um erro do tipo "memory leak"
2  que NÃO ão identificado pela ferramenta Infer - falso negativo*/
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  typedef struct st{
8      int value;
9  }st;
10
11 st * create(int value){
12     st *new = malloc(sizeof(*new));
13     if(new == NULL) return NULL;
14     new->value = value;
15     return new;
16 }
17
18 int main(){
19     st *new = create(5);
20     printf("Value:%d\n",new->value);
21     return 0;
22 }

```

2 ← Memory is allocated →  
 3 ← Assuming 'new' is not equal to null →  
 4 ← Taking false branch →  
 5 ← Calling 'create' →  
 6 ← Returned allocated memory →  
 7 ← Potential leak of memory pointed to by 'new'

**Listing 1.8.** 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));
    if(new == NULL) return NULL;
    new->value = value;
    return new;
}

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

```

**Listing 1.9.** Exemplo mínimo de um erro do tipo fuga de memória identificado pelo Infer.

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

typedef struct st{
    int value;
}st;

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

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

**Listing 1.10.** Relatório de resultados da ferramenta Infer para o exemplo mínimo da Listagem [1.9](#)

```
Found 1 issue

memory_leak_true_positive_infer.c:21: error: MEMORY_LEAK
memory dynamically allocated to 'new' by call to
'create()' at line 19, column 15 is not reachable
after line 21, column 5.
19.     st *new = create(5);
20.     if(new){
21. >   printf("Value:%d\n",new->value);
22.     }
23.     return 0;

Summary of the reports

MEMORY_LEAK: 1
```

**Listing 1.11.** Relatório de resultados da ferramenta Predator para o exemplo mínimo da Listagem 1.9

```
memory_leak_true_positive_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]
```

**Figura 5.** Relatório de erros devolvido pela ferramenta Clang Static Analyzer para o exemplo mínimo da Listagem 1.9

```
1  /*Exemplo mínimo de um erro do tipo "memory leak"
2  que ão identificado pela ferramenta Infer - true positive*/
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  typedef struct st{
8  int value;
9  }st;
10
11 st * create(int value){
12 st *new = malloc(sizeof(st));
13
14     2 ← Memory is allocated →
15
16 if (new == NULL) return NULL;
17
18     3 ← Assuming 'new' is not equal to null →
19
20     4 ← Taking false branch →
21
22 new -> value = value;
23 return new;
24 }
25
26 int main(){
27 st *new = create(5);
28
29     5 ← Returned allocated memory →
30
31 printf("Value:%d\n", new->value);
32
33     6 ← Potential leak of memory pointed to by 'new'
34
35 return 0;
36 }
```

**Estrutura com apontadores.** Na Listagem 1.12 está representado o exemplo mínimo de um erro do tipo fuga de memória que é identificado por todas as ferramentas (consultar a Listagem 1.13 a Listagem 1.14 e a Listagem 1.15), exceto pelo Clang Static Analyzer. Na Listagem 1.16 a fuga de memória foi corrigida e, portanto, não é reportada por nenhuma das ferramentas.

**Listing 1.12.** 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));
    if(!new) return -1;
    new->value = malloc(sizeof(int));
    free(new);
    return 0;
}
```

**Listing 1.13.** Relatório de erros devolvido pela ferramenta Cppcheck para o exemplo mínimo da Listagem [1.12](#)

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

**Listing 1.14.** Relatório de erros devolvido pela ferramenta Infer para o exemplo mínimo da Listagem [1.12](#)

```
Found 1 issue

memory_leak_true_positive_structwithpointer.c:19:
error: MEMORY_LEAK memory dynamically allocated to
'new->value' by call to 'malloc()' at line 14,
column 15 is not reachable after line 19, column 2.
17.                                     return -1;
18.                                     }*/
19. >                                 free(new);
20.                                     return 0;

Summary of the reports

MEMORY_LEAK: 1
```

**Listing 1.15.** Relatório de erros devolvido pela ferramenta Predator para o exemplo mínimo da Listagem 1.12.

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

**Listing 1.16.** Exemplo mínimo de um erro do tipo fuga de memória numa estrutura com apontadores corrigido.

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

typedef struct st{
    int *value;
}st;

int main(){
    st *new = malloc(sizeof(st));
    if(!new) return -1;
    new->value = malloc(sizeof(int));
    free(new->value);
    free(new);
    return 0;
}
```

**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 1.17) 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 1.18 e na Listagem 1.19 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 1.20 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.

**Listing 1.17.** Exemplo mínimo de um erro do tipo fuga de memória utilizando uma função `realloc`.

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

typedef struct st{
    int value;
}st;

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

**Listing 1.18.** Relatório de erros devolvido pela ferramenta Cppcheck para o exemplo mínimo da Listagem 1.17

```
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
```

**Listing 1.19.** Relatório de erros devolvido pela ferramenta Infer para o exemplo mínimo da Listagem 1.17

```
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
```

**Listing 1.20.** 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;
}
```

### E.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 1.21 apenas foram obtidos relatórios de resultados do Infer (Listagem 1.22). Na Listagem 1.23 o erro é corrigido e não deixa de ser reportado pelo Infer.

**Listing 1.21.** 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;
}

```

**Listing 1.22.** Relatório de erros devolvido pela ferramenta Infer para o exemplo mínimo da Listagem [1.21](#)

```

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

```



**Listing 1.23.** Exemplo mínimo de um erro do tipo desreferência nula corrigido.

```
#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;
}

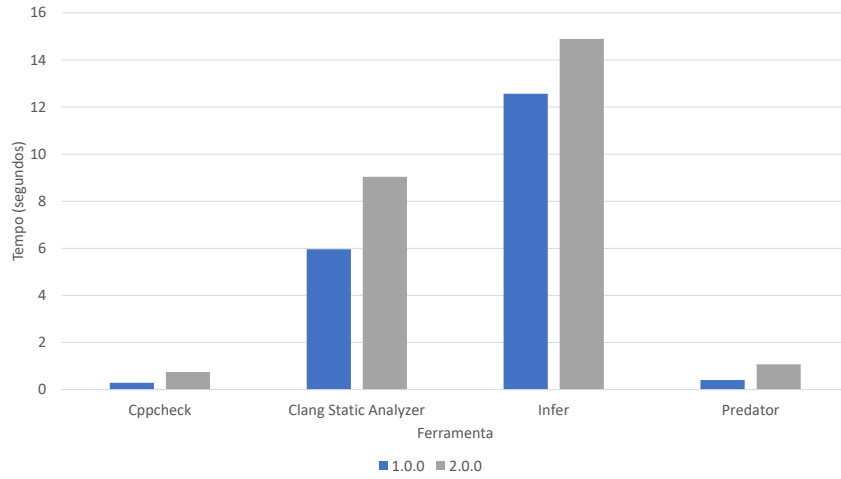
int main(){
    st *new = create(5);
    free(new);
    return 0;
}
```

## F Síntese dos relatórios de resultados das ferramentas

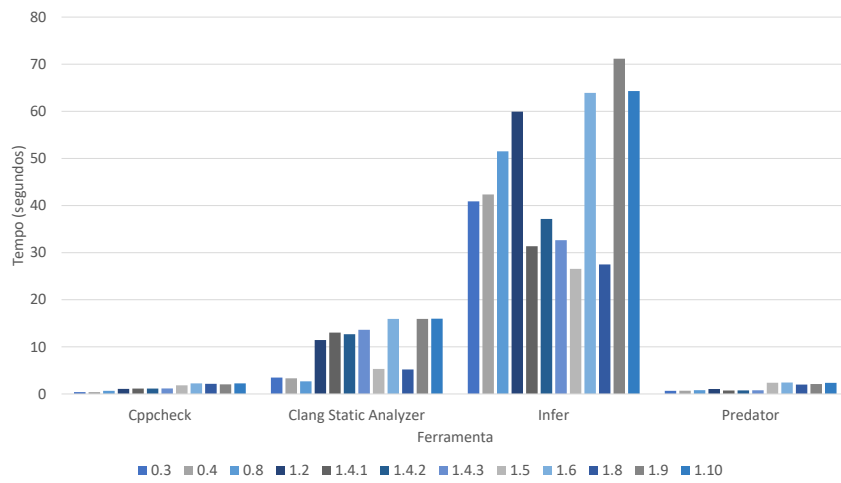
**Tabela 9.** Erros identificados no SDS.

Tipo de erro	Cppcheck		Clang		Static An.		Infer		Predator		Não identificados p/ ferramentas	Identificados apenas p/ uma ferramenta	Total
	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP			
Memory leak	0	1	0	0	0	0	3	0	4	1(Cppcheck)	8		
Null dereference	0	0	0	2	0	4	1	0	1	4(Infer) + 2(CSA)	8		
Invalid dereference	0	0	0	0	0	0	4	0	1	0	4		
Invalid free	0	0	2	0	0	0	1	0	0	0	2		
Dead store	0	0	1	0	1	1	0	0	0	0	2		
Out of bounds	0	0	0	0	0	0	4	0	0	0	4		

## G Tempos de execução das ferramentas



**Figura 6.** Tempos de execução em segundos das ferramentas na análise do SDS.



**Figura 7.** Tempos de execução em segundos das ferramentas na análise do Beanstalkd.

**Tabela 10.** Erros identificados no Beanstalkd.

Tipo de erro	Cppcheck		Clang Static An.		Infer		Predator		Não identificados p/ ferramentas	Identificados apenas p/ uma ferramenta	Total
	FP	TP	FP	TP	FP	TP	FP	TP			
Memory leak	0	1	0	0	4	1	3	2	7	1(Infer) + 1(Predator)	17
Resource leak	0	1	0	0	2	2	0	0	0	2(Infer) + 1(Cppcheck)	5
Null dereference	0	0	0	3	0	9	1	0	0	9(Infer) + 3(CSA)	13
Invalid dereference	0	0	0	0	0	0	125	0	0	0	125
Invalid free	0	0	0	0	0	0	1	0	3	0	4
Dead store	0	0	0	3	8	4	0	0	0	1(Infer)	12
Buffer overflow	0	0	0	0	0	0	0	0	1	0	1
Dangling pointer	0	0	0	0	0	0	0	0	1	0	1

**Tabela 11.** Tempos de execução em segundos das ferramentas na análise do SDS.

Versão	Cppcheck	Clang	Infer	Predator
1.0.0	0,292	5,961	12,562	0,409
2.0.0	0,75	9,037	14,891	1,075

## H Tempo de permanência dos erros nos softwares.

**Tabela 12.** Tempos de execução em segundos das ferramentas na análise do Beantalkd.

	<b>Versão Cppcheck Clang Infer Predator</b>			
0.3	0,397	3,489	40,881	0,673
0.4	0,401	3,311	42,349	0,682
0.8	0,657	2,683	51,502	0,81
1.2	1,075	11,453	59,923	1,06
1.4.1	1,152	13,027	31,358	0,727
1.4.2	1,14	12,685	37,149	0,761
1.4.3	1,179	13,631	32,641	0,773
1.5	1,831	5,323	26,556	2,379
1.6	2,241	15,923	63,914	2,421
1.8	2,144	5,194	27,499	2
1.9	2,031	15,93	71,174	2,116
1.10	2,255	15,982	64,306	2,365

**Tabela 13.** Datas e tempo em meses decorrido desde a introdução até à correção dos erros no SDS.

<b>Erro</b>	<b>Introduzido Corrigido</b>		<b>Intervalo de tempo</b>
sds.c:159: error: memory leak	06/02/2014	25/11/2014	9
sds.c:160: error: null dereference	06/02/2014	25/11/2014	9
sds.c:817: error: null dereference	06/02/2014	x	x
sds.c:891: null dereference	06/02/2014	25/07/2015	17
sds.c: 894: null dereference	06/02/2014	25/07/2015	17
sds.c:92: error: null dereference	25/07/2015	07/02/2018	30
sds.c:1123: error: null dereference	25/07/2015	x	x
sds.c:1240: dead store	25/07/2015	x	x

**Tabela 14.** Datas e tempo em meses decorrido desde a introdução até à correção dos erros no Beanstalkd.

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