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] e do Mars Climate Orbiter

[1] https://around.com/ariane.html

(1999) [1], o bloqueio do terminal 5 no Aeroporto de Londres-Heathrow (2008)[2] e, mais recentemente, o erro encontrado pela Amazon no Jedis (2018)[3].

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 [2], 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 [3], análise automática (dinâmica [4] ou estática [5]) e semi-automática com provadores de teoremas [6]. 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 industria. 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 [7]. 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[4] [8], Clang Static Analyzer[5], Infer[6] [9] e Predator[7] [10]. 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-

[2] `http://www.computerweekly.com/news/2240086013/British-Airways-reveals-what-went-wrong-with-Terminal-5`
[3] `https://github.com/xetorthio/jedis/issues/1747`
[4] `https://github.com/danmar/cppcheck`
[5] `http://clang.llvm.org/`
[6] `https://github.com/facebook/infer`
[7] `https://github.com/kdudka/predator`

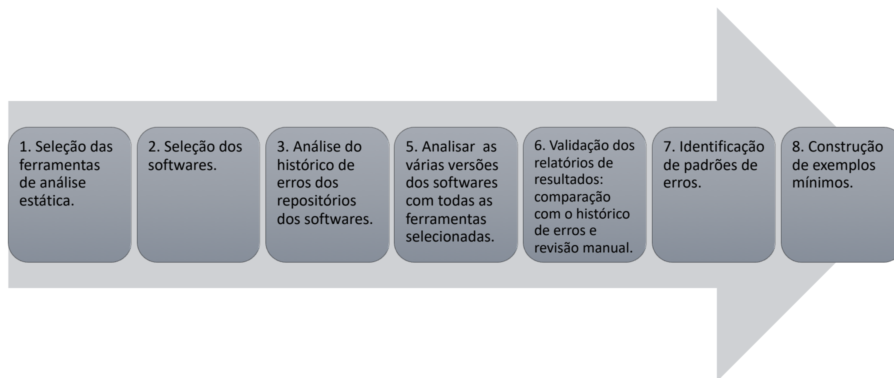


Figura 1. Etapas seguidas durante a realização da experiência.

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 respetivos 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	Fuga de memória	Desreferências inválidas	Operações 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 (listadas na Tabela 5, Apêndice A, da versão completa deste artigo [11]), 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 da versão completa deste artigo. 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 `main` e apenas o Predator requere anotações

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

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 [8], Clang Static Analyzer[8], Frama-C [12], Infer [9], Predator [10] e UNO [13]. 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, selecionámos 16 projetos escritos nas linguagens C/C++, que foram classificados e ordenados de 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*.

[8] `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)[9] biblioteca de *strings* dinâmicas, projetada para aumentar as funcionalidades limitadas das *strings* da biblioteca C;

Beanstalkd[10] gestor de tarefas para aplicações distribuídas;

Tmux[11] multiplexador de terminal, que permite que uma série de terminais possam ser acedidos e controlados a partir de um único terminal;

Memcached[12] 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.

[9] `https://github.com/antirez/sds`
[10] `https://github.com/kr/beanstalkd`
[11] `https://github.com/tmux/tmux`
[12] `https://github.com/memcached/memcached`

Tabela 4. Características dos programas.

Software	Prioridade	Dimensão	Popularidade	Versões	Erros de memória	Erros causam falha	Casos (*issues*) sobre erros de memória	Manipulação de memória	
			(# Estrelas)		(%) total erros	(%) erros mem.	(%) total casos	# Operações	# Apontadores
Simple Dynamic String	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	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 6 e 9 do Apêndice C da versão completa deste artigo. 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 Łforam omitidos os 135 falsos positivos identificados pelo Predator. 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 encontrámos na última versão e que reportámos são:

- *sds.c:1123: error: null dereference*, identificado pelo Infer[13];
- *sds.c:1240: error: dead store*, identificado pelo Clang e pelo Infer[14].

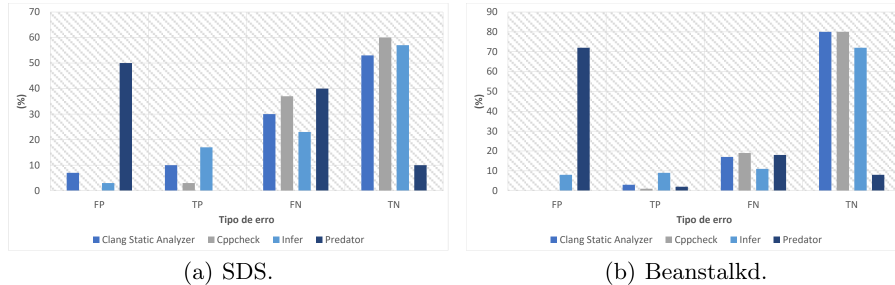
Na Figura 2(a) está representado um gráfico com as percentagens dos vários tipos de erros identificados pelas ferramentas no SDS. Este software retorna, intencionalmente, apontadores para o meio de blocos de memória alocados com `malloc`. Portanto, as operações de desreferência ou libertação de memória 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` [10]). 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 que o Infer não consiga reportar erros que poderia identificar é a existência de

[13] `https://github.com/antirez/sds/issues/99`
[14] `https://github.com/antirez/sds/issues/100`



(a) SDS. (b) Beanstalkd.

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

um tempo limite para execução da análise, que é por vezes atingido antes da análise chegar ao fim [9], 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 [10], 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[15];
- *testheap.c:222: error: memory leak*, identificado pelo Cppcheck e Predator[16].

[15] https://github.com/kr/beanstalkd/issues/384
[16] 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 G da versão completa deste artigo, 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 as 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 H da versão completa deste artigo: 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 I da versão completa deste artigo: 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 do Apêndice D da versão completa deste artigo 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 reportados 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[17]. Atualmente, este software usa duas ferramentas, o Cppchecke o Coverity [14]. 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% [15]. 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 [16].

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

1. Stephenson, A.G., et al.: Mars Climate Orbiter Mishap Investigation Board Phase I Report November 10, 1999. Technical report, NASA (1999)
2. Regehr, J.: A Guide to Undefined Behavior in C and C++, Part 1 – Embedded in Academia. https://blog.regehr.org/archives/213 (2010)
3. Gee, T.: What to Look for in a CodeReview. JetBrains Technical Series (2016)
4. Ball, T.: The concept of dynamic analysis. SIGSOFT Softw. Eng. Notes **24**(6) (1999)
5. Wichmann, B.A., et al.: Industrial perspective on static analysis. Software Engineering Journal **10**(2) (1995)
6. Duffy, D.A.: Principles of Automated Theorem Proving. John Wiley & Sons (1991)
7. Patton, R.: Software testing. Pearson Education India (2006)
8. Marjamaki, D.: Cppcheck 1.81. (2017) http://cppcheck.sourceforge.net/.
9. Calcagno, C., Distefano, D.: Infer: An Automatic Program Verifier for Memory Safety of C Programs. In: NASA Formal Methods International Symposium. Volume 6617 of LNCS., Springer (2011)

[17] https://github.com/LibreOffice/core

10. Dudka, K., Peringer, P., Vojnar, T.: Byte-Precise Verification of Low-Level List Manipulation. In: International Static Analysis Symposium. Volume 7935 of LNCS., Springer (2013)
11. Monteiro, P., Lourenço, J., Ravara, A.: Uma análise comparativa de ferramentas de análise estática para deteção de erros de memória (2018) http://arxiv.org/abs/1807.08015.
12. Correnson, L., et al.: Frama-C User Manual. (2017) https://frama-c.com/download.html.
13. Holzmann, G.J.: Static source code checking for user-defined properties. In: Integrated Design and Process Technology. Volume 2. (2002)
14. Llaguno, M.: 2017 coverity scan report. Technical report, Synopsys (2017) https://www.synopsys.com/blogs/software-security/2017-coverity-scan-report-open-source-security/.
15. Project, C.: Coverity Scan – Static Analysis. https://scan.coverity.com/projects/211
16. Project, L.: Cppcheck - HTML report – LibreOffice 2018-01-27. https://dev-builds.libreoffice.org/cppcheck_reports/master/index.html