UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Bruno Evangelista Costa

Monitoramento do Envelhecimento de Software Relacionado à Memória: Um Estudo Exploratório

Uberlândia, Brasil

2018

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Bruno Evangelista Costa

Monitoramento do Envelhecimento de Software Relacionado à Memória: Um Estudo Exploratório

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Rivalino Matias Jr.

Universidade Federal de Uberlândia – UFU

Faculdade de Ciência da Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2018

Bruno Evangelista Costa

Monitoramento do Envelhecimento de Software Relacionado à Memória: Um Estudo Exploratório

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Rivalino Matias Jr.
Orientador

Anilton Joaquim da Silva

Autran Macêdo

Uberlândia, Brasil
2018

Agradecimentos

Quero agradecer, em primeiro lugar, a Deus, pela saúde e força para superar esta longa caminhada. Agradeço também a minha família, pelo incentivo e apoio incondicional. Por fim, gostaria de agradecer a todos os professores que me acompanharam durante a graduação, em especial ao Prof. Dr. Rivalino Matias Jr., pela paciência, suporte e ensinamentos durante todos estes anos de orientação.

"Eduquem as crianças e não será necessário castigar os homens."

Resumo

Os efeitos acumulados do envelhecimento de software têm influência direta na taxa de falhas de software relacionadas a este fenômeno. Até então, os efeitos de envelhecimento de software mais investigados são os relacionados ao uso da memória principal, como problemas de vazamento e fragmentação de memória. Neste trabalho, apresenta-se um conjunto prático de conhecimento, teórico e prático, para auxiliar no entendimento sólido de um dos mais importantes problemas na monitoração dos efeitos do envelhecimento de software relacionados à memória, com foco de vazamento de memória. Discute-se importantes riscos de usar indicadores de envelhecimento de sistema e de aplicação bem conhecidos, assim como propor soluções efetivas para ambos os casos.

Palavras-chave: envelhecimento, software, vazamento, memória, indicadores, monitoramento.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

SO/OS	Sistema operacional
SNMP	Protocolo simples de gerenciamento de rede
PFM	Módulo de gerenciamento proativo de falhas
VM	Máquina virtual
VMM	Monitor de VM
JVM	Máquina virtual java
RSS	Tamanho do *working set* do processo
VIRT	Tamanho do processo no espaço de endereçamento da memória virtual
KB	Kilobyte
MB	Megabyte
GB	Gigabyte

Sumário

1 Introdução

Nos últimos vinte e três anos, o fenômeno de envelhecimento de software vem sendo sistematicamente investigado e reconhecido como uma importante ameaça para se alcançar sistemas de computação confiáveis. Envelhecimento de software é definido como uma degradação gradual do estado interno do software durante sua vida operacional (Y. Huang, C. Kintala, N. Kolettis and N. Fulton, 1995). Envelhecimento em um sistema de software, assim como nos seres humanos, é o resultado de um processo acumulativo (R. Matias, P. A. Barbetta, K. S. Trivedi and P. J. F. Filho, Mar. 2010). Os efeitos acumulados de ocorrências sucessivas de erros relacionados ao envelhecimento de software influenciam diretamente a taxa de falhas. Portanto, os efeitos do envelhecimento de software são as consequências dos erros acumulados causados por falhas relacionadas a este problema. A ocorrência destas falhas leva gradualmente o estado interno do sistema para um estado errôneo (M. Grottke, R. Matias, and K. Trivedi, 2008). Esta mudança gradual, que é consequência da acumulação dos efeitos do envelhecimento, é a natureza fundamental do fenômeno de envelhecimento de software. Devido a esta característica acumulativa, um pequeno tempo de execução geralmente não é suficiente para detectar sinais de envelhecimento de software em um determinado sistema. Por isso, este fenômeno é usualmente observado em sistemas de software que rodam continuamente e que são executados durante um longo período. Problemas como inconsistência de dados, erros numéricos e esgotamento dos recursos do sistema operacional (SO) são exemplos dos efeitos do envelhecimento de software. Embora tenham sido reportadas várias classes diferentes de efeitos de envelhecimento de software (ex. (S. Garg, A. van Moorsel, K. S. Trivedi and K. S. Trivedi, 1998)-(J.C.T. Araujo, R. Matos, P.R.M. Maciel, R. Matias, and I. Beicker, 2011)), os efeitos relacionados à degradação dos recursos do SO, particularmente aqueles relacionados à memória principal, são os mais citados na literatura (ex. (M. Grottke, R. Matias, and K. Trivedi, 2008), (R. Matias and P. J. Freitas Filho, 2006)-(J. Alonso, J.L. Berral, R. Gavaldà, and J. Torres, Mar. 2010)). Fundamentalmente, efeitos de envelhecimento de software relacionados à memória principal são causados por problemas de vazamento e fragmentação (A. Macedo, T.F. Borges and, R. Matias, 2010). Vazamento de memória (Q. Ni, W. Sun, and S. Ma, 2008) é uma falta de software bem conhecida, a qual é principalmente causada pelo uso incorreto das rotinas de gerenciamento de memória (uso desbalanceado de malloc e free, por exemplo). Por outro lado, a fragmentação de memória (M.S. Johnstone and P.R. Wilson, 2008) é uma consequência direta da dinâmica de execução do sistema, que pode se tornar uma grande preocupação para aplicações que necessitam de grandes e fisicamente contíguas alocações de memória.

Muitos trabalhos experimentais sobre envelhecimento e rejuvenescimento de soft-

ware têm focado em caracterizar o fenômeno de envelhecimento e avaliar a eficiência de mecanismos de rejuvenescimento. Para ambos os casos, detectar a existência de envelhecimento é essencial, que depende de monitorar os efeitos de envelhecimento. Neste trabalho, são discutidos importantes aspectos do monitoramento dos efeitos do envelhecimento de software relacionado à memória, especialmente em relação ao vazamento de memória. Note que tais aspectos não foram considerados em trabalhos anteriores, embora sua grande importância na prevenção de conclusões errôneas ao diagnosticar a presença de envelhecimento de software em um dado sistema. O entendimento correto da fenomenologia de envelhecimento de software é um requisito importante para fazer decisões acertadas e fornecer diagnósticos confiáveis. Por isso, o objetivo é apresentar um corpo prático de conhecimento para dar suporte à compreensão sólida dos aspectos mais importantes do monitoramento dos efeitos de envelhecimento relacionados ao vazamento de memória. Como um estudo exploratório, este trabalho discute achados que irão ajudar pesquisadores a escolher melhores estratégias para medir os efeitos do envelhecimento. Para fazer isso, foram utilizados experimentos controlados em nosso estudo. As próximas seções deste trabalho estão organizadas da seguinte forma: Serão apresentados os mecanismos fundamentais do vazamento de memória. Este conhecimento é necessário para compreender as outras seções. Após isso, serão apontadas as descobertas recentes em relação às desvantagens de usar indicadores de envelhecimento conhecidos para monitorar os efeitos de envelhecimento relacionado à memória. Finalmente, serão mostradas as considerações finais.

2 Revisão Bibliográfica

2.1 Vazamento de Memória e Envelhecimento de Software

Embora a fragmentação de memória tenha ganhado mais atenção na literatura recente sobre envelhecimento de software (ex. (J.C.T. Araujo, R. Matos, P.R.M. Maciel, R. Matias, and I. Beicker, 2011), (R. Matias, I. Beicker, B. Leitão, and P.R.M. Maciel, 2010), (R. Matias and P. J. Freitas Filho, 2006)), a maioria dos trabalhos de pesquisa experimental nesta área tem focado no vazamento de memória. A detecção e mensuração correta de vazamentos de memória não são tarefas triviais. Isto exige um conhecimento aprofundado dos mecanismos fundamentais por trás deste problema. Um vazamento de memória ocorre quando o processo da aplicação que aloca blocos de memória dinamicamente e, por alguma razão, não libera esta memória (integralmente ou parcialmente) de volta para o sistema operacional durante sua execução (A. Macedo, T.F. Borges and, R. Matias, 2010). Existem várias razões para que isto ocorra, e propôs-se a classificá-las em dois tipos principais: retenção de memória involuntária e retenção de memória voluntária. Retenção de memória involuntária ocorre quando um processo é incapaz de liberar o bloco de memória alocado previamente. Isso acontece principalmente pelo motivo da perda da referência (endereço) do bloco alocado. Por exemplo, sobrescrevendo o valor da variável ponteiro que guarda o endereço do bloco alocado. Neste caso, não é possível liberar o bloco dado que seu endereço foi sobrescrito, ou seja, perdido, o que resulta em um bloco órfão (Q. Ni, W. Sun, and S. Ma, 2008). Outro exemplo é o uso desbalanceado das rotinas malloc (ou new) e free (ou delete). Neste caso, não há uma chamada da free() correspondente para cada chamada da malloc(), fazendo com que alguns blocos de memória erroneamente não sejam liberados. Geralmente, problemas involuntários são causados por erros (bugs) de software. Diferentemente, a retenção de memória voluntária não é consequência de falhas de software, mas é relacionada ao projeto do software. Por exemplo, para melhorar o desempenho da aplicação, um processo pré-aloca porções de memória para atender as solicitações que serão realizadas. Pode-se encontrar este padrão de projeto em muitos softwares servidores, onde a proposta principal é evitar o custo computacional de operações repetitivas para a alocação e liberação de objetos de memória. Alguns algoritmos intencionalmente nunca liberam os recursos alocados, mantendo eles para que sejam reutilizados futuramente durante toda a execução da aplicação. Em alguns casos, a quantidade de recursos alocados aumenta monotonicamente devido a picos esporádicos na carga de trabalho. Este padrão de uso de memória não é incomum na maioria das aplicações atuais. De forma genérica, a literatura sobre gerenciamento de memória se referência a este problema como memory blowup (T.B. Ferreira, R. Matias, A.

Macedo and L.B. Araujo, 2011). Em todos os cenários supracitados, o resultado prático é o consumo de memória gradual pelo processo da aplicação durante sua execução. Em execuções de longo prazo, isso pode esgotar a memória principal e assim causar falhas de sistema devido ao colapso (ex. OS trashing (U. Vahalia, 1995)) ou mal desempenho do mesmo. Isto é importante para correlacionar os casos de retenção de memória voluntária e involuntária com o papel dos alocadores de memória (A. Macedo, T.F. Borges and, R. Matias, 2010), (U. Vahalia, 1995). Em ambos os casos, os blocos que "vazaram" são parte da *heap* do processo. Uma vez que o vazamento de memória acelera a saturação da *heap*, blocos de memória adicionais devem ser requisitados para o sistema operacional. Portanto, quando a *heap* está saturada (cheia), uma nova área de memória (nova *heap*) é adicionada ao espaço de endereçamento do processo, fazendo com que o tamanho residente do processo cresça. Baseado neste fato, pode-se tirar uma importante conclusão para o estudo: Enquanto a *heap* do processo não é ampliada, o tamanho do processo continua inalterado mesmo sob a ocorrência de vazamentos de memória.

2.2 Trabalhos Relacionados

O trabalho de (S. Garg, A. van Moorsel, K. S. Trivedi and K. S. Trivedi, 1998) teve como objetivo propor uma metodologia para detecção e estimação de envelhecimento no sistema operacional UNIX. Para isso, foram apresentados o *design* e a implementação de uma ferramenta baseada no protocolo simples de gerenciamento de rede, que foi usada para coletar o uso dos recursos e os dados de atividade do sistema. A partir disto, técnicas estatísticas foram aplicadas a estes dados com a finalidade de detectar ou validar a existência de envelhecimento. Para qualificar os efeitos do envelhecimento nos recursos do sistema, foi proposto um "tempo estimado até a exaustão", calculado a partir de técnicas bem conhecidas de estimativa de inclinação.

O protocolo simples de gerenciamento de rede (SNMP) é um protocolo de aplicação que oferece serviços de gerenciamento de rede no conjunto de protocolos Internet. Segundo o trabalho, existem três constituintes essenciais para uma ferramenta baseada no SNMP, que são:

1. O módulo de gerenciamento proativo de falhas (PFM), responsável por definir os objetos utilizados para determinar a "saúde"de um ambiente de trabalho UNIX.

2. O agente PFM, que tem seu processo executado em segundo plano no ambiente monitorado e fica responsável por "escutar"em um número de porta informado. Ao receber uma requisição do tipo get, o agente executa algumas instruções para obter o valor do objeto-folha requisitado do sistema operacional.

3. Gerenciador de PFM, responsável por obter os valores dos objetos desejados. Para isso, envia uma requisição do tipo get para os agentes remotos.

Para obter estes dados, a ideia principal foi monitorar e coletar periodicamente dados que se referem à "saúde"do software em execução. A ferramenta de coleção de dados baseada no SNMP descrita anteriormente foi implantada em seis estações de trabalho UNIX heterogêneas que foram conectadas em uma rede local. Os dados foram coletados a cada 15 minutos, sendo guardados em um arquivo especificado pelo usuário em um formato X-Y, onde X é o nome do objeto e Y seu valor. O processamento destes dados foi feito fora da ferramenta responsável pela coleta. Em caso de ocorrência de erro, uma mensagem de "Sem Resposta"era gravada. O tempo limite de espera era de dois minutos, e quando ocorria algum erro em alguma das máquinas, ela era reiniciada juntamente com o processo de coleta. Para quantificar o envelhecimento no UNIX e obter as respostas desejadas, foram utilizadas sugestões visuais e técnicas de análise de séries temporais clássicas, como análise de dependência linear e análise de dependência periódica, além de detecção de tendências e estimativas.

Como resultado do estudo, baseado na metodologia criada para estimar e detectar envelhecimento em um software operacional, foi proposta uma métrica, "Tempo estimado para a exaustão", para cada um dos recursos quanto ao envelhecimento. Quanto maior esta métrica, menor é o efeito do envelhecimento no recurso em questão. Esta métrica ajuda na comparação do efeito de envelhecimento em diferentes recursos do sistema e também na identificação de recursos importantes para monitorar e gerenciar. Segundo o artigo esta métrica pode vir a ser uma ajuda para prever a ocorrência de falhas relacionadas ao envelhecimento, e também pode ajudar no desenvolvimento de estratégias para softwares tolerantes a falhas.

Muitos estudos foram conduzidos para entender o fenômeno do envelhecimento de software. O trabalho de (R. Matias and P. J. Freitas Filho, 2006) tem como objetivo apresentar os resultados de um trabalho de pesquisa experimental, para avaliar os efeitos do envelhecimento em um servidor web, assim como a performance de um agente de rejuvenescimento. Sua maior contribuição é a identificação e validação analítica dos fatores que contribuem para o envelhecimento do servidor web. Para isso, foi adotado o design de técnica de experimento (DOE) para caracterizar este fenômeno.

Um "agente de rejuvenescimento"(SRA) foi implementado e integrado em no ambiente do servidor para reduzir os efeitos do envelhecimento. Com este ambiente, o tamanho dos processos httpd puderam ser mantidos sob controle e com maior disponibilidade, além de prover maior performance para o servidor web. A primeira parte do estudo foi verificar se o servidor web estava apresentando sinais de envelhecimento e, caso haja sinais positivos, determinar os fatores e seus graus de influência em um servidor web. Durante os testes de capacidade do sistema, vários recursos foram monitorados. Foi verificado que a maioria da degradação do servidor ocorria na memória física, assim, considerou-se que o envelhecimento do Apache foi o responsável por este problema.

Existem duas aproximações para a implementação do agente de rejuvenescimento, definidas como *open-loop* e *closed-loop*. Na *open-loop*, o agente é programado para executar o mecanismo de rejuvenescimento independente do status do sistema, sendo executado em intervalos predeterminados. Na aproximação *closed-loop*, o agente monitora o sistema com a finalidade de obter dados que seriam úteis para determinar a melhor técnica e o melhor cronograma para o rejuvenescimento. Estes monitoramentos podem ser feitos no modo *on-line* ou no modo *off-line*. No modo *on-line*, o monitoramento e a análise são feitas durante a execução do agente, já no modo *off-line*, estas atividades continuam após a execução do mesmo. Como o foco do estudo era na validação do mecanismo de rejuvenescimento contra os efeitos do envelhecimento, a aproximação *closed-loop* no modo *on-line* foi escolhida.

O monitoramento do tamanho dos processos httpd foi definido como a política de rejuvenescimento, assumido um valor limiar de 450 MB. A execução do mecanismo de rejuvenescimento ocorria quando o tamanho total dos processos httpd ultrapassavam este valor. Foram utilizadas duas formas de rejuvenescimento: total e parcial. No rejuvenescimento total, o sistema é completamente reinicializado, o que amplia o tempo de inatividade no serviço. Na parcial, apenas uma parte do sistema era reinicializado e o período de inatividade depende de como os componentes reinicializados são organizados.

Os resultados dos experimentos iniciais mostraram que os fatores 'page type' e 'page size' foram responsáveis por mais de 99% da variação do tamanho da memória nos processos httpd, tanto individualmente quanto combinadas. Baseado nestes resultados, um SRA foi implementado seguindo a abordagem *closed-loop*. Neste estágio, a maior dificuldade encontrada foi a definição do mecanismo de rejuvenescimento que causaria menor impacto na disponibilidade do serviço e evitaria que o código httpd sofresse alterações. Como resultado, o SRA demonstrou boa performance no controle do tamanho dos processos, evitando problemas encontrados em testes sem rejuvenescimento. Outro benefício foi a manutenção da taxa de resposta do servidor, uma vez que esta taxa foi afetada negativamente pelo envelhecimento. O principal foco deste trabalho foi nas causas do envelhecimento, uma vez que estes aspectos não foram considerados em pesquisas anteriores, e a sua principal contribuição foi a identificação e validação analítica dos fatores que contribuem para os efeitos do envelhecimento de software.

Segundo (A. Macedo, T.F. Borges and, R. Matias, 2010), o envelhecimento de software é um fenômeno definido como uma degradação continua de sistemas de software durante sua execução, sendo que seus efeitos na memória é um dos problemas mais importantes nesta área de pesquisa. Sendo assim, compreender suas causas e como funcionam é um requisito fundamental para a criação de sistemas de software confiáveis. Este trabalho foca em dois problemas de memória que causam envelhecimento: vazamento e fragmentação, além de explicar os mecanismos dos efeitos do envelhecimento de software relacionado

à memória. Juntamente com a explicação, um dos objetivos é apresentar um estudo experimental que ilustra como o vazamento e a fragmentação ocorrem e como acumulam com o passar do tempo gerando falhas relacionadas ao envelhecimento no sistema.

O alocador de memória é responsável por implementar operações para o gerenciamento da pilha de memória. Para manipular a memória, o alocador utiliza funções como *malloc()*, *realloc()* e *free()*, que são implementadas como parte do alocador, que pode requisitar por mais memória quando não há memória suficiente disponível para atender a requisição. Esta memória adicional fica então ligada à pilha do processo e gerenciada pelo seu alocador de memória. As operações de alocação podem existir em dois níveis: nível de *kernel* e nível do usuário. Estruturalmente, não há diferença entre os alocadores criados para manipular a memória nestes dois níveis, pois seus principais componentes são basicamente os mesmos. Os metadados do alocador são usados para manter controle da memória alocada e liberada, e as operações são utilizadas para controlar estas áreas.

Para o estudo experimental, foram executados dois experimentos: Experimento de fragmentação de memória e experimento de vazamento de memória. O propósito é mostrar como estes dois efeitos do envelhecimento relacionados à memória ocorrem internamente, mostrando os mecanismos destes efeitos dentro dos alocadores de memória. Foram criados dois programas (Mfrag e Mleak) para implementar os casos de teste relacionados à fragmentação e ao vazamento de memória respectivamente. No experimento de fragmentação de memória, o Mfrag consome controladamente toda a pilha de memória e depois libera vários blocos de uma maneira não contígua, o que fragmenta a pilha, e, após isso, solicita por um novo bloco de memória. Como nenhum dos blocos disponíveis (liberados) é era grande o suficiente para satisfazer a requisição, é possível ver o alocador emitir uma requisição ao sistema operacional por novas áreas de memória para o processo (sbrk()). O monitoramento da execução do programa foi feito usando o strace, um rastreador que grava chamadas de sistema feitas por um processo. Foi possível observar que quando o programa requisitou 8 KB, o srbk() foi chamado embora houvessem 64.440 *bytes* livres na pilha. A nova área de memória foi solicitada porque a pilha estava sofrendo de fragmentação de memória.

O vazamento de memória ocorre quando uma quantidade de memória alocada anteriormente não é liberada e não pode ser usada novamente, normalmente por motivo de perda de referência. Em nível de usuário, seus efeitos acabam no processo da aplicação quando o processo termina, assim, a memória atribuída ao processo retorna para o sistema operacional. Processos sendo executados há muito tempo podem causar o crescimento do mesmo devido a vazamentos sucessivos, reduzindo a disponibilidade de memória do sistema. No experimento, o programa Mleak causava vazamento de memória intencionalmente. O resultado deste processo foi observado ao monitorar a quantidade de memória residente do processo (RSS). O monitoramento foi feito através do uso do

arquivo /proc/pid/status que guarda informações úteis de um dado processo em execução identificado pelo seu pid, sendo possível observar um aumento do tamanho do processo na memória principal.

O vazamento de memória é um efeito do envelhecimento bem conhecido, e um dos problemas mais discutidos na literatura. Porém este foi o primeiro trabalho relacionado ao envelhecimento que discutiu fragmentação de memória na prática. Como resultado do trabalho, foi possível verificar que dois problemas surgem com a fragmentação. O primeiro é o aumento do tamanho do processo devido a novas requisições de memória até mesmo quando há memória suficiente disponível (mas não de uma maneira contínua). O segundo problema é sobre a performance, pois cada requisição necessita entrar no modo *kernel* e depois retornar ao modo usuário, o que pode impactar de forma significativa a performance do processo.

Para que os mecanismos de rejuvenescimento possam ser aplicados de maneira eficiente, é muito importante identificar as causas mais significantes do envelhecimento de software. Sendo assim, (R. Matias, I. Beicker, B. Leitão, and P.R.M. Maciel, 2010) tem como objetivo apresentar um estudo que explora técnicas de instrumentação do *kernel* para medir os efeitos do envelhecimento de software. Estas instrumentações estão presentes em praticamente todos sistemas operacionais modernos, e o trabalho visa mostrar como este mecanismo pode ser usado para monitorar indicadores de envelhecimento específicos de aplicação ou de sistema.

Os indicadores de sistema fornecem informações relacionadas a subsistemas que são compartilhados (ex. OS e VM) e, portanto, são influenciados por outros elementos do sistema. Indicadores desta categoria são normalmente utilizados para avaliar os efeitos de envelhecimento no sistema como um todo. Indicadores específicos de aplicação fornecem informação do processo de uma aplicação específica, assim, a informação é mais precisa se comparada à informação fornecida pelos indicadores de sistema. Para estes dois tipos de indicadores de envelhecimento, é possível coletar dados a nível de usuário e a nível de *kernel*.

Recursos de rastreamento do *kernel* do Linux são relativamente novos. Atualmente existem dois *frameworks* disponíveis, o kprobes e o ftrace. Juntamente a estes *frameworks*, programas rastreadores foram desenvolvidos com a finalidade de permitir que usuários comuns utilizem a instrumentação do *kernel* de uma maneira mais fácil. O *SystemTap* é um programa para o Linux que foca em coletar informação *on-line* do ambiente *kernel* e do ambiente de usuário. Desta maneira o usuário pode ter uma visão mais aprofundada do que está ocorrendo no *kernel* do sistema. A principal ideia deste programa é nomear os eventos para que o usuário consiga toda informação requisitada quando um determinado evento ocorre. Um evento pode ser o nome de uma função, uma linha de código específica, uma exceção ou uma interrupção, o que significa que quando algum destes eventos ocorrem,

os dados são coletados.

Foram planejados dois tipos de experimentos: vazamento de memória e fragmentação de memória. O objetivo foi ilustrar como a instrumentação do *kernel* pode ser usada para medir os efeitos relacionados ao envelhecimento. Os efeitos escolhidos são relacionados a problemas de memória. Estes problemas podem ocorrer em dois ambientes: ambiente do usuário e ambiente do *kernel*.

Ao fim dos experimentos, foi possível mostrar como as modernas infraestruturas de instrumentação do sistema operacional foram usadas para medir o efeito do envelhecimento no software. Medir e analisar os efeitos do envelhecimento desta nova maneira é um passo muito importante em direção ao melhor entendimento do fenômeno do envelhecimento de software. Este recurso abre novas possibilidades na pesquisa sobre envelhecimento assim como técnicas de ressonância magnética abriram para a pesquisa sobre o cérebro, permitindo que cientistas vissem em "tempo de execução", o que acontece no cérebro humano quando este está sobre algum estímulo específico.

Sistemas de software que são executados continuamente por um longo período de tempo sempre sofrem envelhecimento de software, uma degradação progressiva causada por falhas. No trabalho de (R. Matias, A. Andrzejak, F. Machida, D. Elias and K Trivedi, 2014) foi proposta uma aproximação sistemática com a finalidade de detectar o envelhecimento de software que tem o menor tempo de teste e a maior precisão se comparado a uma detecção via testes de estresse e detecção de tendências. A abordagem baseia-se numa análise diferencial em que uma versão de software sob teste é comparada contra uma versão anterior em termos de mudanças de comportamento de métricas de recursos, comparando os dados por meio de um gráfico de divergência. O foco do estudo foi na detecção e na avaliação do vazamento de memória e na avaliação dos gráficos de divergência.

Remover completamente bugs relacionados ao envelhecimento na fase de desenvolvimento de um software é muito difícil e pode ser até inviável. Mas técnicas podem ser utilizadas para melhorar a confiabilidade do software, apesar delas não serem perfeitas para remover todos as falhas relacionadas ao envelhecimento. Um dos objetivos deste artigo é a detecção de envelhecimento de software em um menor período de teste. Para isso, a aproximação projetada se trata de um teste de sistema baseado na análise de software diferencial, por exemplo, uma comparação entre uma versão nova do software em teste e sua versão estável anterior que passou no teste e, portanto, é considerada uma versão robusta do software. Uma vez que o cálculo do gráfico de divergência é aplicável a qualquer tipo de sinal a partir de dados de séries temporais, a abordagem proposta pode ser combinada com diferentes técnicas de análise de regressão dos dados (por exemplo, linear, médias móveis, e técnicas de controle estatístico). Ao comparar vários gráficos divergência uns contra os outros, foi possível identificar o método mais eficaz para ser usado para a detecção rápida de envelhecimento para uma dada situação.

São descritos dois aspectos chaves a serem considerados para qualquer abordagem de detecção de vazamento de memória *on-line*, que são as métricas do sistema que será monitorado e os valores de limiar para decidir de forma confiável a existência de vazamento de memória.

A abordagem para detectar envelhecimento de software é composta em três passos. Primeiramente foram realizadas medições em uma versão de software em teste. Após isso, os dados coletados foram tratados para fins de análise estática, e por fim foi possível fazer a detecção de padrões de uso de recursos inesperados. A terceira etapa é baseada na comparação dos dados recolhidos no primeiro passo em relação aos dados obtidos a partir de uma versão estável anterior do software em teste. Uma vez que esta abordagem é independente de métricas do sistema e técnicas de processamento de dados, se trata detecção de envelhecimento software robusta, que pode ser aplicada a várias combinações de métricas e técnicas de uma maneira sistemática.

Os resultados mostraram que o método proposto atinge um bom desempenho para a detecção de vazamentos de memória em comparação com técnicas amplamente adotadas em trabalhos anteriores. A aproximação utilizada é suficientemente genérica para suportar várias técnicas de processamento de sinal combinadas com diferentes números e tipos de métricas de sistema. A introdução dos gráficos de divergência podem ajudar engenheiros a realizar análises visuais. Outro achado importante deste estudo é relacionado à comparação entre memória residente do processo e o uso da pilha de memória. Uma vez que o uso da pilha de memória gerou melhores resultados para todos os cenários com baixa taxa de manifestação de vazamento de memória, esta métrica é recomendada especialmente para planos de teste onde o software apresenta baixas taxas de falhas relacionadas a vazamento de memória.

Segundo estudo de (F. Machida, A. Andrzejak, R. Matias and E. Vicente,), o envelhecimento de software é de difícil detecção devido ao longo tempo até que ele se manifeste durante a execução do programa. Portanto, detectar rapidamente e de forma precisa o envelhecimento é importante para eliminar os efeitos deste fenômeno desde o desenvolvimento do software. Sabendo disso, o objetivo deste trabalho foi avaliar se o teste Mann-Kendall é uma aproximação efetiva para detectar envelhecimento de software. Esta técnica busca por tendências monotônicas em séries temporais, e estudos sobre envelhecimento de software muitas vezes consideram a existência de tendências em certas métricas como indicativo de existência deste fenômeno.

No contexto de detecção de envelhecimento, séries temporais são vestígios de métricas relevantes do sistema computacional assim como o tamanho do conjunto residente ou uso do *heap* de memória. Como muitos tipos de envelhecimento se manifestam através do aumento do uso de recursos, inclinações positivas destes indicadores são comumente interpretadas como sinais de envelhecimento de software. Portanto, estudos experimentais

de envelhecimento software têm aplicado o teste de Mann-Kendall, juntamente com tais indicadores com o propósito de detectar efeitos de envelhecimento.

O principal problema desta abordagem, é que a observação que detectou a tendência pode não ser causado por envelhecimento de software. Se um indicador possui normalmente uma grande variação, este teste pode produzir vários alarmes falsos. Para amenizar a variância dos valores do indicador de envelhecimento, a quantidade de dados submetidos para o teste é aumentada.

O experimento teve seu foco na detecção de envelhecimento de software causada pelo vazamento de memória. Para os experimentos controlados, foi criado um gerador de carga de trabalho que emula o comportamento de uma aplicação de uso geral, que repetidamente requisita e libera blocos de memória em intervalos de tempo aleatórios. Este gerador foi programado na linguagem C, e usa as funções malloc e free. Um erro foi inserido propositalmente durante a liberação de memória, resultando em vazamento de memória. Os indicadores de envelhecimento utilizados neste experimento foram a memória livre, a memória residente do processo e o uso do *heap* de memória.

Os experimentos mostraram que o teste usando a memória livre não foi capaz de distinguir casos de vazamento de memória dos casos onde este fenômeno não ocorria. O desempenho da detecção melhorou consideravelmente ao usar a memória residente e o uso do *heap* de memória, sendo que o primeiro se apresentou um pouco menos sensível e específico em alguns casos. O teste de Mann-Kendall com limitação de dados não é uma técnica confiável para detecção de envelhecimento, pois produz vários falsos alarmes em casos onde o vazamento não ocorria e falsos negativos onde o vazamento ocorria. Quando este teste é utilizado sem limitações, passa a ser mais adequado para detecção de envelhecimento, mas necessita de mais dados para chegar a um alto nível de confiabilidade, além de depender da qualidade do indicador de envelhecimento utilizado. O estudo também mostrou que utilizar métricas básicas do sistema como indicador de envelhecimento tem impacto significante na capacidade de detecção. Utilizar a memória livre não foi útil no contexto do experimento executado, mas por outro lado, utilizar a memória residente do processo e o uso do *heap* de memória fez com que os resultados fossem satisfatórios.

Tendo em vista que aspectos conceituais da base deste fenômeno não foram abrangidos na literatura. O artigo (M. Grottke, R. Matias, and K. Trivedi, 2008) tem como objetivo discutir aspectos fundamentais do fenômeno do envelhecimento de software, além de introduzir novos conceitos, interligando-os ao conjunto de conhecimentos atuais com a finalidade de compor uma taxonomia base para o estudo sobre o envelhecimento de software.

Para isso, primeiramente é explicada a física das falhas de software. Cada erro é causado pela ativação de uma falta, e a sua propagação leva o software para um estado defeituoso. Isso é importante para explicar o processo de envelhecimento de software,

uma vez que uma característica comum deste fenômeno é o fato de que a taxa de falhas aumenta juntamente com o tempo de execução do processo.

Segundo este trabalho, muitos defeitos relacionados ao envelhecimento de software são consequências de faltas de software. A maioria das faltas relacionadas ao envelhecimento são causadas devido ao acúmulo de erros desta natureza dentro do estado interno do sistema. Portanto, é a acumulação de erros que leva o estado interno do sistema para um estado defeituoso. Um erro ocorre quando os padrões de ativação de defeitos, ou seja, os fatores ou as combinações de fatores que ativam o defeito, se unem a um defeito de software.

Outro conceito mostrado no artigo trata-se da volatilidade dos efeitos do envelhecimento. Estes efeitos são considerados voláteis se podem ser removidos com a reinicialização do sistema ou processo afetado, enquanto são denominados não voláteis quando persistem após a reinicialização. Desta maneira, fenômenos como vazamento e fragmentação de memória são considerados efeitos de envelhecimento voláteis, enquanto fragmentação do sistema de arquivos e dos metadados de um banco de dados são considerados efeitos não voláteis. Efeitos de envelhecimento em um sistema em execução só podem ser detectados através do monitoramento de indicadores de envelhecimento, que conseguem mostrar se o sistema está saudável ou não.

Ao final, foi possível criar um exemplo de como uma falha relacionada ao envelhecimento pode ser analisada baseado nos conceitos discutidos no trabalho. Com isso, foram propostas algumas características do fenômeno do envelhecimento de software, tais como:

- O efeito do envelhecimento somente é reversível com alguma intervenção externa. No melhor cenário, os erros poderão parar de acumular durante períodos em que o sistema não estiver exposto a fatores de envelhecimento.

- O efeito do envelhecimento somente depende do tempo de clock se este tempo de clock constitui o ambiente interno do sistema que influencia a acumulação ou propagação de erros.

-O tempo da CPU somente influencia o efeito do envelhecimento se ativar faltas relacionadas a este fenômeno ou se os causar naturalmente durante seu tempo de execução. Pode influenciar também se o tempo da CPU for parte constituinte do ambiente interno do sistema que gera acumulação de erros ou propagação dos mesmos.

3 Estudo Experimental

3.1 Monitorando os Efeitos de Envelhecimento Relacionados ao Vazamento de Memória

Detectar e mitigar os efeitos do envelhecimento são tarefas essenciais para conseguir alta confiabilidade em sistemas que sofrem deste problema. Os efeitos do envelhecimento de software podem ser detectados somente durante a execução do sistema, por meio do monitoramento de indicadores de envelhecimento e comparando-os com padrões de bom funcionamento estabelecidos para o sistema alvo.

3.1.1 Indicadores de envelhecimento de software

Conceitualmente, indicadores de envelhecimento de software (M. Grottke, R. Matias, and K. Trivedi, 2008) são indicadores para detecção de envelhecimento assim como antígenos são indicadores para a detecção de câncer. Indicadores de envelhecimento são variáveis que representam um estado estável do sistema monitorado.

Comparar os valores monitorados destas variáveis com os valores padrão de um dado sistema pode ajudar a revelar a presença dos efeitos do envelhecimento. Estas variáveis explicativas podem ser utilizadas individualmente ou combinadas para mostrar se o sistema monitorado é considerado "saudável" ou não em termos de envelhecimento de software. Um estado saudável significa que o estado interno está estável, sendo o oposto de um estado de provável falha por esgotamento de recursos (envelhecimento). Até então, a escolha de um conjunto otimizado de indicadores de envelhecimento para usar na detecção de envelhecimento de software se mantém um problema em aberto.

A qualidade dos indicadores escolhidos possui efeito direto na eficácia dos mecanismos de rejuvenescimento (R. Matias and P. J. Freitas Filho, 2006), (W. Xie, Y. Hong, and K. S. Trivedi, 2005), (T. Dohi, K. Goseva-Popstojanova, and K.S. Trivedi, 2000), dado que sua eficiência depende do tempo de ativação que é influenciado pela precisão dos indicadores de envelhecimento escolhidos. Indicadores de envelhecimento podem ser propostos em diferentes níveis do sistema, tais como componentes da aplicação, processos, middleware, sistema operacional, máquina virtual (VM), monitor de VM (VMM), e assim por diante. Dado que o envelhecimento de software pode ocorrer em cada um destes níveis, é importante adotar os indicadores apropriados para cada camada. Em geral, indicadores de envelhecimento são classificados em duas categorias de acordo com a sua granularidade (M. Grottke, R. Matias, and K. Trivedi, 2008): indicadores em nível de sistema e

indicadores específicos de aplicação. Indicadores de envelhecimento em nível de sistema fornecem informação relacionada a subsistemas que são compartilhados e, portanto, influenciados por outros componentes do sistema. Exemplos de subsistemas compartilhados são: Middleware de aplicação, sistema operacional, VM e monitor de VM. Indicadores desta categoria são sempre usados para avaliar os efeitos do envelhecimento no sistema como um todo (*system wide*). Exemplos de indicadores de sistema largamente utilizados são a quantidade de memória física livre/usada, tamanho da área de *swap* livre/usada, número de arquivos abertos, número de descritores (ex. *sockets*) em uso, entre outros.

Indicadores de envelhecimento específicos de aplicação fornecem informação específica sobre um processo individual. Exemplos de indicadores de envelhecimento amplamente utilizados nesta categoria são o tamanho do *working set* (RSS) do processo, tamanho da pilha da JVM e o tempo de resposta d aplicação. Quando o processo da aplicação de interesse não está rodando diretamente sob o sistema operacional, mas em uma plataforma de máquina virtual, como programas Java e C, estes indicadores de envelhecimento podem ser aplicados indiretamente, direcionados ao processo que executa a máquina virtual (ex., processo java).

Para ambas as classes de indicadores de envelhecimento (de sistema e de aplicação), é possível coletar dados em nível de usuário e em nível de *kernel* do SO. A coleta de dados em nível de usuário é implementada por programas de propósito especial que são executados no espaço do usuário (R. Matias and P. J. Freitas Filho, 2006), (Y. Bao, X. Sun, and K.S. Trivedi, 2005). Note que neste nível o monitoramento do sistema é limitado às informações que o sistema operacional fornece sobre o processo. Por exemplo, no Linux o sistema de arquivos virtual /proc disponibiliza várias informações úteis para este fim.

Alternativamente, é possível coletar dados em nível de *kernel* por meio de mecanismos de instrumentação de *kernel* (R. Matias, I. Beicker, B. Leitão, and P.R.M. Maciel, 2010), (D. Controneo, R. Natella, and S. Russo, 2010). Neste caso, uma vez que a medição ocorre dentro do *kernel*, teoricamente qualquer informação estará disponível tanto para indicadores de sistema quanto para indicadores de aplicação. Embora seja poderosa, esta abordagem requer mudança no código do *kernel*, estaticamente ou dinamicamente.

No primeiro método, o código fonte do *kernel* deve ser modificado e recompilado para acomodar as rotinas de monitoramento que irão extrair os dados de interesse (D. Controneo, R. Natella, and S. Russo, 2010). Uma desvantagem deste método é que isso só é possível quando o código fonte do sistema operacional está disponível. No segundo método, as mudanças dentro do código do *kernel* ocorrem em tempo de execução, onde as rotinas de monitoramento são dinamicamente ligadas ao *kernel* (R. Matias, I. Beicker, B. Leitão, and P.R.M. Maciel, 2010); para isso não é necessário o código fonte do sistema operacional. Isto é possível porque o sistema operacional implementa facilidades de instrumentação específicas. Estas facilidades permitem que o pesquisador crie scripts/rotinas e as ligue

ao *kernel* sem mudar o seu código fonte. Normalmente, estas rotinas são ativadas por eventos específicos do *kernel* (ex. uma chamada de sistema ou seu retorno). Todos sistemas operacionais modernos oferecem mecanismos de instrumentação. Por exemplo, Detours no MS Windows (G. Hunt and D. Brubacher, 1999), Dtrace no UNIX Solaris (B. Cantrill, M. Shapiro, and A. Leventhal, 2004) e *SystemTap* no Linux (B. Jacob, P. Larson, B. Leitão, and S.A.M.M Da Silva, 2008) são infraestruturas de instrumentação de *kernel*.

Como foi discutido nas seções anteriores, os efeitos do envelhecimento relacionado à memória, especialmente vazamentos de memória, são os mais investigados na literature sobre envelhecimento e rejuvenescimento de software. Em seguida, são apresentadas questões importantes, que devem ser levadas em consideração durante o monitoramento do vazamento de memória para propósito de detecção de envelhecimento de software, dado que negligenciar tais questões pode levar ao diagnóstico errôneo sobre a existência ou ausência dos efeitos de envelhecimento de software.

3.1.2 Usando indicadores de envelhecimento de sistema

Esta categoria de indicador de envelhecimento é adequada quando não há conhecimento prévio do sistema investigado, pois oferece uma primeira visão geral dos padrões de uso de recursos. Indicadores de envelhecimento de sistema são adequados para executar experimentos de triagem, o que permite ao pesquisador classificar e selecionar as variáveis de interesse. Isto é particularmente importante quando se trabalha com um sistema novo ou desconhecido e muitas variáveis candidatas são consideradas preliminarmente.

A maioria das pesquisas anteriores sobre envelhecimento e rejuvenescimento de software tem usado esta abordagem. Devido à natureza compartilhada dos indicadores de sistema, eles usualmente apresentam ruídos significativos. Especificamente na investigação de vazamento de memória, os indicadores de envelhecimento de sistema mais frequentemente utilizados têm sido a memória física livre/usada e a área de *swap*. Por uma questão de simplicidade, será considerada apenas a memória física na discussão. Monitorando apenas estes indicadores é muito difícil fazer um diagnóstico preciso se o sistema sofre ou não de vazamento de memória. Para ilustrar esta dificuldade, foi conduzido um experimento no qual executamos uma carga de trabalho que simula operações típicas de um sistema de arquivos em um servidor web. Para isso, utilizou-se o filebench (Filebench,) que é um gerador de carga de trabalho para este propósito, o qual configuramos para produzir três tipos de cargas: baixa (w1), moderada (w2), e alta (w3). A Tabela 1 mostra as diferenças entre os três perfis de carga.

Tabela 1 – Perfis de Carga de Trabalho

Perfil	Parâmetros de Carga de Trabalho
w1	#arquivos=5000; #threads=25
w2	#arquivos=10000; #threads=50
w3	#arquivos=20000; #threads=100

Estes três perfis de carga são executados duas vezes de uma maneira cíclica, onde cada ciclo dura aproximadamente uma hora. O tempo total de cada execução do experimento é de seis horas, e repetimos cada execução 10 vezes para reduzir as influências de erros experimentais. Assim, os resultados discutidos são baseados nos valores médios. A Figura 1 mostra os valores monitorados obtidos durante o experimento 1. Os valores do eixo y representam a memória total utilizada, e os valores do eixo x representam o tempo em intervalos de 30 segundos (6h no total, ou 720 intervalos).

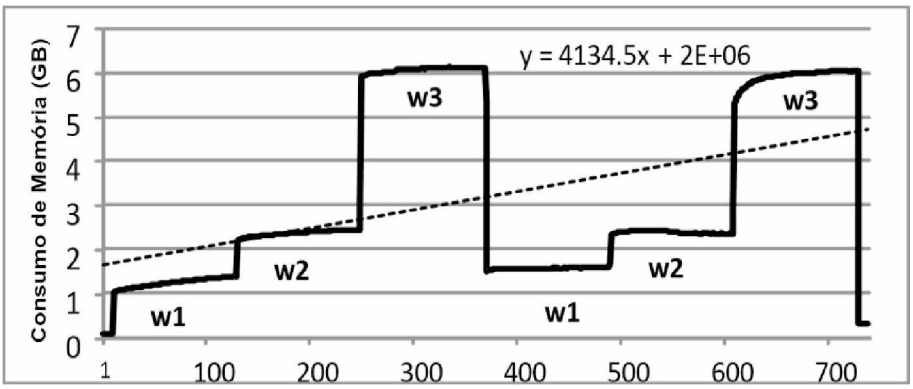


Figura 1 – Padrão típico do incremento do uso de memória observado em cenários com vazamento.

Como se pode observar, os resultados claramente indicam uma tendência de aumento no consumo de memória. Este padrão é frequentemente observado em muitos resultados experimentais reportados em pesquisas sobre envelhecimento e rejuvenescimento de software que utilizam indicadores de envelhecimento de sistema (ex., memória física utilizada). Entretando, analizando cuidadosamente os dados, notamos que essa tendência não é consequência do envelhecimento de software, mas simplesmente o efeito do cache de disco do SO (também chamado de *buffer-cache*). Como as cargas de trabalho adotadas (w1, w2, w3) são de I/O de disco intensas, o gerenciamento de memória do *kernel* do SO (T.B. Ferreira, R. Matias, A. Macedo and L.B. Araujo, 2011) requisita memória física inutilizada para o alocador do nível de páginas com a finalidade de criar objetos de cache de disco adicionais. Este comportamento é esperado considerando que: i) exista memória física livre disponível a nível de usuário, e ii) o disco do subsistema de I/O esteja sobre pressão. Neste cenário, não é incomum observar a quantidade de memória

livre decair consideravelmente. Este comportamento pode ser mal interpretado facilmente como vazamento de memória, quando na verdade temos a memória disponível sendo temporariamente transferida para o subsistema *buffer-cache* do SO; note que esta memória pode ficar disponível para o nível de usuário novamente assim que ela for requisitada. Sendo assim, observar somente a memória física livre/usada ou em conjunto com outros indicadores de envelhecimento pouco precisos (ex., aréa de *swap* livre/usada) pode levar a conclusões errôneas sobre a existência de vazamento de memória em um sistema.

Para este caso, com o intuito de melhorar o diagnóstico, sugere-se monitorar não somente a quantidade de memória usada ou disponível, mas também comparar os dados com a memória utilizada pelo *buffer-cache* junto com a quantidade total de memória alocada no nível de usuário. Estas análises combinadas ajudarão a entender o fluxo de memória dentro do sistema investigado, assim como enxergar se vazamentos estão realmente ocorrento, especialmente no espaço do usuário onde as aplicações são executadas. No geral, esta abordagem conjunta é especialmente importante para evitar erros de diagnóstico ao usar indicadores de sistema relacionados à memória. A Figura 2 mostra o resultado da abordagem proposta. Como pode ser visto, a quantidade de memória utilizada é fortemente influenciada pelo subsistema de *buffer-cache* em vez do subsistema do nível de usuário onde deveria ocorrer o vazamento de memória da aplicação.

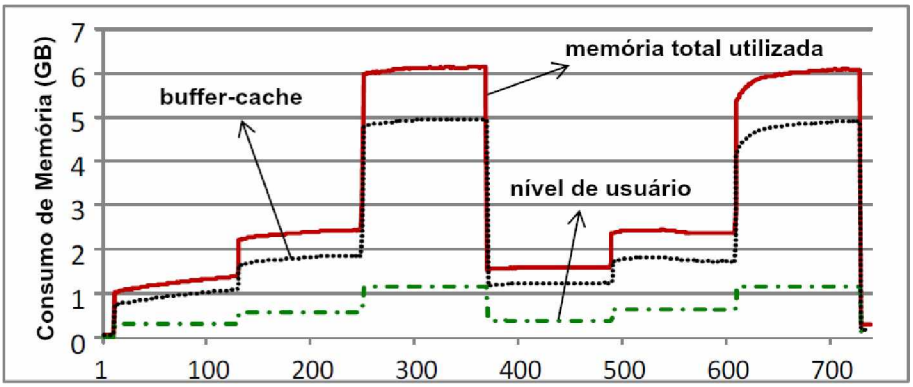


Figura 2 – Monitoramento dos indicadores de envelhecimento de sistema combinados.

Preferencialmente, em sistemas conhecidos, recomendamos comparar os valores dos indicadores monitorados com um ponto de referência livre de envelhecimento. Para ilustrar isto, conduzimos um segundo experimento. No Exp. 2 executamos um processo de uma aplicação juntamente com a carga de trabalho do Exp. 1 rodando em segundo plano. O Exp. 2 foi composto por quatro testes (2.1 - 2.4). No 2.1, monitoramos o memória total utilizada enquanto a aplicação era executada, sem vazamento de memória, em conjunto com a carga de trabalho em segundo plano. No 2.2, injetamos vazamentos de memória dentro do processo da aplicação. O vazamento ocorre de uma maneira aleatória segundo a uma distribuição uniforme (ver Figura 3). O teste 2.3 é simitar ao 2.2, porém injetamos vazamento de memória no nível de *kernel* ao invés do nível de usuário como foi feito no 2.2.

Isto é realizado por meio de um módulo de *kernel* que desenvolvemos para este propósito, que segue o mesmo algoritmo usado para a aplicação com vazamento de memória (ver Figura 3). No teste 2.4, injetamos vazamentos de memória na aplicação e no nível de *kernel*. A Tabela 2 mostra um sumário de todos os testes conduzidos no Exp. 2.

Algorithm 1. No memory leaking

```
t: sleep time in seconds
f: multiple of page size
loop
    t = random (1..30);
    f = random (1..5);
    c = malloc (1024 * f);
    if (c is equal to NULL) then break;
    for each position in c
        c[position] = 0;
    sleeps for t seconds;
    free (c);
end loop
```

Algorithm 2. Memory leaking

```
t: sleep time in seconds
f: multiple of page size
k: luck number
loop
    t = random (1..30);
    f = random (1..5);
    c = malloc (1024 * f);
    if (c is equal to NULL) then break;
    for each position in c
        c[position] = 0;
    sleeps for t seconds;
    k = random (even..odd);
    if (k is even) then free (c);
end loop
```

Figura 3 – Algorítimos utilizados no experimento 2.

Tabela 2 – Perfis dos testes do experimento 2.

Teste	Perfil	Resultados
2.1	Sem vazamento de memória (baseline)	Fig. 4(a)
2.2	Vazamento de memória na aplicação	Fig. 4(b)
2.3	Vazamento de memória no kernel do SO	Fig. 4(c)
2.4	Vazamento de memória no kernel do SO e na App	Fig. 4(d)

A Figura 4 mostra os resultados do Exp. 2. O teste 2.1 é o nosso ponto de referência sem envelhecimento, que é usado na comparação com todos os outros resultados. Como pode ser visto, todos os testes do Exp. 2 mostram resultados similares em termos de tendência de aumento de memória. Isto ocorre porque a carga de trabalho em segundo plano força o efeito de *caching* do disco do SO, discutido na Sessão 2.1.2, que pratica-

mente domina a variabilidade do uso de memória, escondendo a maioria dos efeitos de envelhecimento injetados tanto no nível de usuário quanto no nível de *kernel*.

No geral, o tamanho predominante de alocações de memória, em média, é menor do que 64 *bytes* (T.B. Ferreira, R. Matias, A. Macedo and L.B. Araujo, 2011), que é facilmente ofuscado pela maior variabilidade das muitas cargas de trabalho comuns, ex., no Exp. 2 a quantidade média de memória vazada é de 200 KB em seis horas.

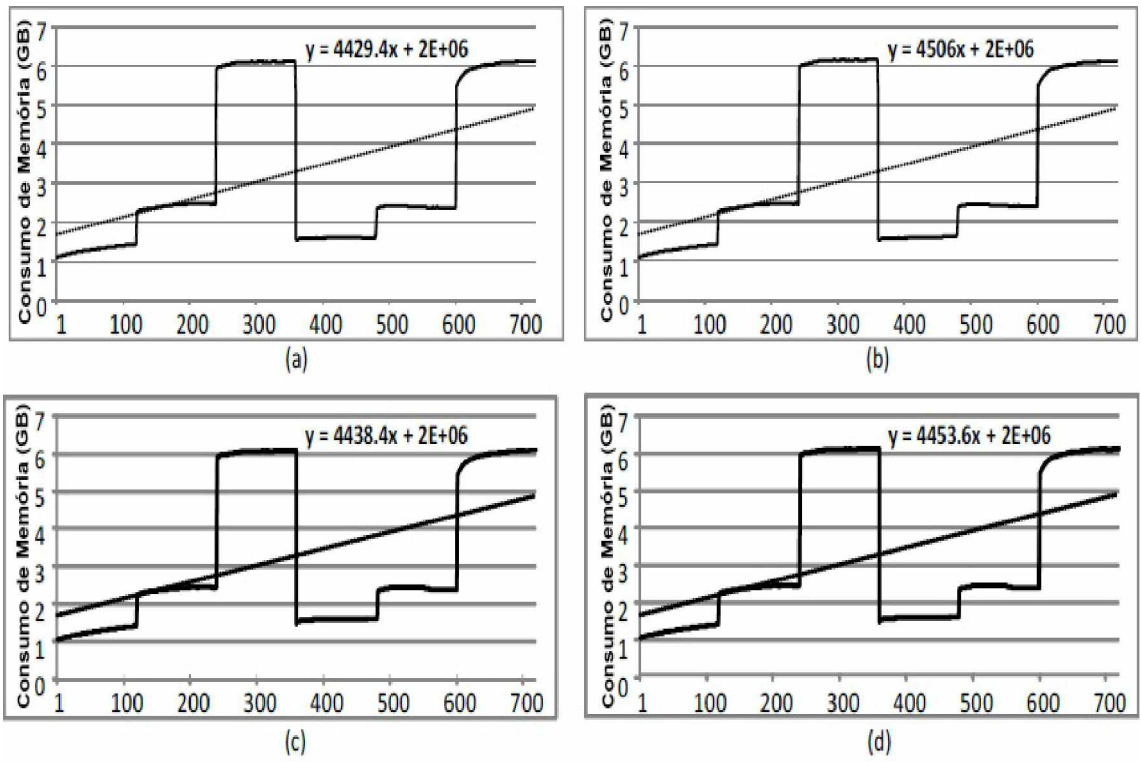


Figura 4 – Resultados do experimento 2.

Adotar indicadores de referência (*baselines*) pode reduzir as desvantagens de usar indicadores de envelhecimento de sistema. Por exemplo, comparando os resultados obtidos em 2.1 (referência) com outros três testes (2.2, 2.3, e 2.4), pode se observar o aumento na taxa de consumo causada por vazamentos de memória injetados propositalmente para esse experimento. Embora a injeção de vazamento de memória em nível de usuário e de *kernel* tenham seguido o mesmo algoritmo, a taxa de consumo no nível do *kernel* é menor do que no nível de usuário. Isso acontece por que o alocador de memória do *kernel* (A. Macedo, T.F. Borges and, R. Matias, 2010), neste caso o slab (U. Vahalia, 1995), trabalha baseado em blocos de recurso, o que reduziu os sinais de vazamento de memória durante o experimento 2. Baseado nestes resultados, é possível concluir que indicadores de envelhecimento de sistema sofrem de muitas desvantagens. Notamos que muitas pesquisas anteriores em envelhecimento e rejuvenescimento de software basearam suas análises apenas em indicadores de envelhecimento de sistema.

Para estes casos, recomendamos fortemente adotar uma medida de referência (*baseline*) para reduzir riscos de falso-positivos.

3.1.3 Usando Indicadores de Envelhecimento de Aplicação

Para evitar equívoco nos valores obtidos com indicadores de sistema, os indicadores de aplicação são uma melhor alternativa para oferecer informação de um processo individual da aplicação. Especialmente para detectar vazamentos de memória, (R. Matias and P. J. Freitas Filho, 2006) propôs o uso do tamanho do conjunto residente do processo (RSS) como indicador de envelhecimento. Este indicador permite monitorar a quantidade de memória física que um determinado processo de aplicação está usando. Monitorar este indicador é muito mais efetivo do que usar indicadores de sistema como memória física livre ou usada. Entretanto, para utilizá-lo o pesquisador deve saber qual processo deve ser monitorado. Embora este indicador de envelhecimento tenha sido usado e apresentado bons resultados em trabalhos anteriores (ex. (R. Matias, P. A. Barbetta, K. S. Trivedi and P. J. F. Filho, Mar. 2010), (R. Matias and P. J. Freitas Filho, 2006), (J.C.T. Araujo, R. Matos, P.R.M. Maciel, R. Matias, and I. Beicker, 2011), (A. Macedo, T.F. Borges and, R. Matias, 2010), (R. Matias, I. Beicker, B. Leitão, and P.R.M. Maciel, 2010), (R. Matias and P. J. Freitas Filho, 2006)), observamos recentemente que este indicador também sofre de algumas desvantagens. Investigando os mecanismos de vários alocadores de memória (T.B. Ferreira, R. Matias, A. Macedo and L.B. Araujo, 2011), aprendemos que vazamentos de memória não aumentam imediatamente o tamanho do processo quando ocorrem. É necessário um tempo de propagação do vazamento para que este seja observado de fora da *heap* do processo, uma vez que o tamanho do processo só aumenta quando uma nova área de memória é requisitada ao sistema operacional a fim de estender a *heap* saturada. Sendo assim, neste intervalo de tempo de observação, não se pode detectar ocorrências de vazamento de memória pelo monitoramento do tamanho do conjunto residente do processo (RSS). Para ilustrar este problema, realizamos o experimento 3. Neste experimento, executamos um programa que aloca inicialmente 1024 blocos de 1024 *bytes* (1 MB no total).

Após isso, são liberados 512 blocos e alocamos 512 novos blocos de 1024 *bytes*, mas agora vazando todos eles. A Figura 5 mostra o resultado do experimento 3. Observamos que o RSS do processo aumentou logo após a primeira seção de alocações de memória. Isso ocorre por que a *heap* inicial, criada pelo alocador de memória padrão (ptmallocv2 (T.B. Ferreira, R. Matias, A. Macedo and L.B. Araujo, 2011)), tem apenas 132 KB de tamanho, e requisitamos 1 MB de memória. Então, novas áreas de memória são adicionadas ao espaço de endereçamento do processo, fazendo com que o tamanho do conjunto residente (RSS) aumente. Em seguida, metade da memória alocada é liberada, e isto não altera o valor do RSS do processo.

```
char *p[1024];
main(){
 unsigned long j=0,i=0,l=0;
                                        RSS= 224 kB
 for(i=0; i<1024;i++){
    p[i]=malloc(1024);
    if ( p[i]==NULL)  exit(0);
    for(l=0;l<1024;l++) p[i][l]='*';
 }
                                        RSS= 1276 kB
 for(i=0;i<512;i++)  free(p[i]);
                                        RSS= 1276 kB
 for(i=0; i<512;i++){
    p[0]=malloc(1024);
    if ( p[0]==NULL) exit(0);
    for(l=0;l<1024;l++) p[0][l]='*';
 }
                                        RSS= 1276 kB
}
```

Figura 5 – Resultados do experimento 3.

Este é um achado importante, dado que o senso comum é que logo após a liberação da memória alocada, essa porção de memória irá retornar ao sistema operacional. Isso está acontecendo por que o alocador ptmallocv2 move os blocos liberados para as listas livres da *heap* de modo a mantê-los para reutilização futura (T.B. Ferreira, R. Matias, A. Macedo and L.B. Araujo, 2011). Em seguida, forçamos o vazamento de 512 blocos de 1024 *bytes* e o RSS do processo ainda continua o mesmo. Isto confirma, experimentalmente, nossa hipótese mencionada acima de que nem todo vazamento é capturado imediatamente monitorando apenas o RSS.

Estes achados têm um efeito importante nos mecanismos de rejuvenescimento, dado que várias abordagens de rejuvenescimento esperam o indicador de envelhecimento monitorado (ex. RSS) chegar a um determinado limiar (*threshold*) para serem executadas. Nossos experimentos mostram que um monitoramento preciso de vazamento de memória deve ser realizado dentro da *heap* em vez de fora, como tem sido comumente implementado nos trabalhos anteriores. Não o fazendo, o acompanhamento do RSS pode não revelar precisamente a taxa correta de vazamento de memória, mas somente a taxa em que o processo está aumentando a sua área de *heap*. Note que semelhante aos resultados obtidos no experimento 1 (ver Figura 1), o aumento do tamanho do conjunto residente (RSS)

não deve ser sempre considerado como um sinal de vazamento de memória. Isso pode ser simplesmente a demanda da aplicação por mais memória da *heap*, e não uma consequência de vazamento de memória causado por defeitos do software ou problemas em seu projeto. No caso de aplicações que adotem o padrão de "conjunto de recursos", esta tendência de aumento é muito comum. Então, para verificar se o aumento do RSS do processo deve ser considerado como uma consequência de vazamentos de memória, propomos testar a aplicação em um ambiente controlado, onde é possível reduzir a quantidade de memória física. Isso pode ser facilmente obtido tanto fisicamente quanto por meio de máquinas virtuais. Se o processo continua a crescer por trás dos limites da memória física, isto é, aumentando o tamanho da sua memória virtual, então isso deve ser considerado um problema de vazamento de memória, seja voluntaria ou involuntária (ver Capítulo 3).

Em ambos os casos o crescimento descontrolado do tamanho do processo irá saturar os recursos do sistema. Na situação onde o crescimento do processo não é causado por vazamento de memória, mas simplesmente pela consequência de picos transitórios de cargas de trabalho, o tamanho do processo na memória virtual não deve ser significativamente maior do que o RSS, que é limitado à memória física disponível. Por esse motivo, consideramos que monitorar o RSS em combinação com o tamanho virtual do processo (VIRT) é uma melhor estratégia do que usar apenas o RSS. Note que o RSS representa o tamanho das partes do processo residentes na memória física, e se a memória física disponível é limitada e o processo continua a crescer por trás deste limite, o RSS não pode capturar o aumento deste tamanho. Assim, pode-se concluir que o acompanhamento dos tamanhos combinados de RSS e VIRT oferece um indicador de envelhecimento muito mais preciso para a detecção de vazamento de memória.

4 Combinando Indicadores de Aplicação

4.1 Indicador VIRT

Como dito no capítulo anterior, foi proposto realizar outro experimento com a finalidade de deixar o estudo ainda mais preciso, mas desta vez utilizando outro indicador de envelhecimento, o VIRT. Como já foi explicado anteriormente, o VIRT é o tamanho virtual de um processo, que é a soma da memoria que o processo está realmente usando, a memória que o processo mapeou para si mesmo e a memória dividida com outros processos.

4.1.1 Utilizando o Indicador de Aplicação VIRT

Para analisar melhor o comportamento do tamanho virtual do processo (VIRT), foi realizado um experimento semelhante ao experimento 3 deste artigo. Executamos novamente um programa que aloca 1024 blocos de 1024 *bytes* resultando em 1 MB de memória alocada. Após isso, liberamos 512 blocos e alocamos novamente 512 blocos de 1024 *bytes*. A Figura 6 ilustra os resultados do experimento 4. O comportamento do VIRT foi semelhante ao comportamento do RSS no experimento 3. Novamente observamos que o VIRT do processo aumentou logo após a primeira alocação de memória, mas ao liberarmos metade da memória alocada, o valor da VIRT não se altera.

Como foi dito no experimento 3, ao contrário do que se espera, ao liberar a memória através do comando *free*, ela não é liberada para o sistema no mesmo instante, pois o alocador de memória move os blocos disponíveis para o *heap* com o propósito de utilizá-la novamente no futuro caso seja necessário. Deste modo, o tamanho da memória virtual do processo não se altera, mostrando que tanto o RSS quanto o VIRT se comportam de forma semelhante quando monitorados de fora do *heap* ao invés de dentro. Ambos experimentos (3 e 4) mostram que, para o monitoramento de vazamento de memória ser mais preciso, devemos realiza-lo fora do *heap* de memória da aplicação.

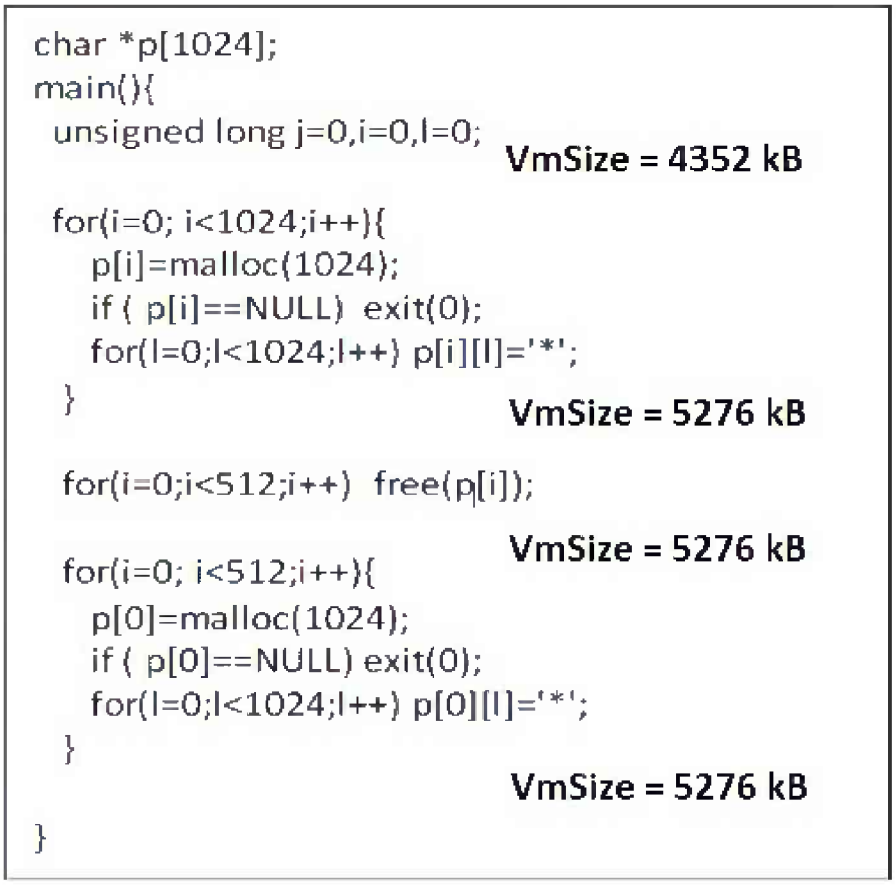


Figura 6 – Resultado do experimento 4.

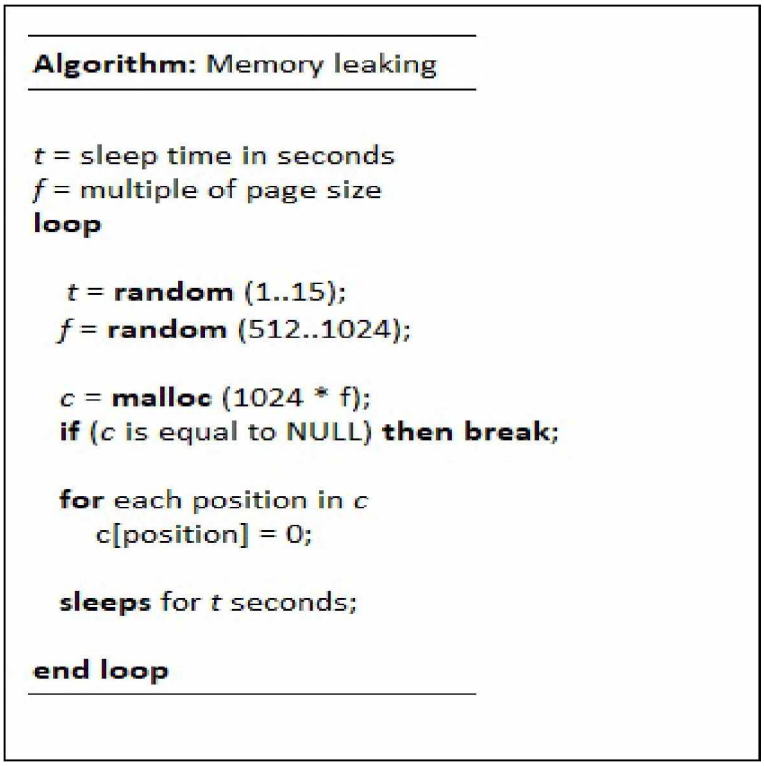
4.2 Experimento Com Ambos Indicadores

4.2.1 Dados do Experimento

Como dito anteriormete, a combinação entre o VIRT e o RSS oferece um indicador de envelhecimento de software mais seguro para constatação de vazamento de memória. Para demonstrar este cenário, nós conduzimos um experimento no qual executamos uma aplicação que gera, propositalmente, vazamento de memória de uma forma aleatória. Esta aplicação foi executada em um ambiente controlado para que fosse possível simular escassez de memória. Para isso, foi utilizada uma máquina virtual com aproximadamente 1.7GB de memória RAM, fazendo com que, após a inicialização do sistema, restassem por volta de 600MB de memória livre para o SO. Desta maneira, foi possível observar como cada um dos indicadores se comportam em um ambiente com limite de memória, já que ambos se comportam de forma muito semelhante quando há memória em abundância.

Como é possivel observar na Figura 7, a aplicação aloca, aleatoriamente, blocos de memória que vão de 512KB a 1MB e, após isso, perde a referência do mesmo para alocar um outro bloco, vazando a memória reservada anteriormente. Cada alocação ocorre em

um período arbitrário que pode ser de 1 a 15 segundos. Após a alocação, preenchemos a memória com caracteres antes de forçar o vazamento.



Algorithm: Memory leaking

```
t = sleep time in seconds
f = multiple of page size
loop

   t = random (1..15);
  f = random (512..1024);

  c = malloc (1024 * f);
  if (c is equal to NULL) then break;

  for each position in c
     c[position] = 0;

  sleeps for t seconds;

end loop
```

Figura 7 – Algoritmo do experimento 5.

Em paralelo ao algoritmo, foi executado um script que ficou responsável de coletar os dados do experimento. A cada minuto eram coletados dados referentes à memória virtual (VIRT) e a memória real (RSS) utilizadas pela aplicação. Cada experimento teve uma duração de sete horas, totalizando em 420 amostras de consumo de memória. Para melhorar a consistência dos dados obtidos, o experimento foi replicado seis vezes.

4.2.2 Resultados do Experimento

Ao final do experimento, foi possível notar com os resultados que a memória real da aplicação para de crescer assim que o sistema chega a um ponto em que a memória física está praticamente esgotada, mas a memória virtual continua a crescer indefinidamente (ver figura 8).

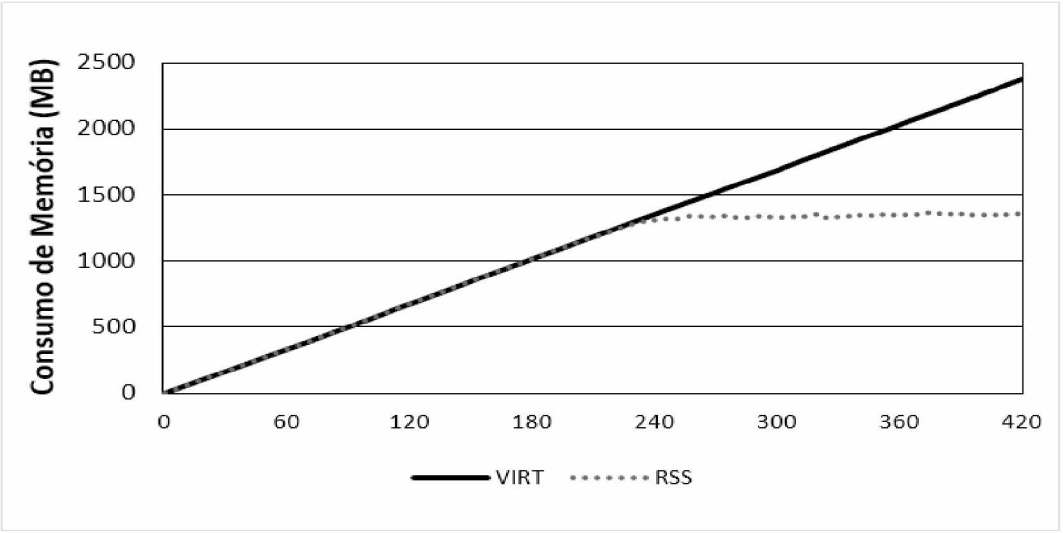


Figura 8 – Resultado do experimento 5.

Esse é um achado significativo, pois mostra que somente observar a memória real da aplicação pode não fornecer insumos suficientes para definir ou não a existência de vazamento de memória no ambiente monitorado. Como foi possível observar na figura 8, o tamanho do conjunto residente do processo (RSS) para de crescer, mas o tamanho do processo continua a aumentar na memória virtual. Isso confirma a suspeita de que observar somente o indicador RSS não é o suficiente para definir, de forma confiável, se há ou não envelhecimento de software na aplicação.

O aumento do tamanho do processo vem da memória virtual utilizada pela aplicação, pois, ao contrário da memória real, que é limitada pelo tamanho da memória disponível pelo sistema, a memória virtual utiliza o disco rígido, que, normalmente, é mais abundante do que a memória RAM. O estudo mostra que, em cenários com escassez de memória, acompanhar somente a memória real da aplicação pode gerar um falso negativo para a existência de envelhecimento de software na aplicação, pois o vazamento de memória continua aumentando o tamanho do processo na memória virtual, mas o aumento da memória real é cessado.

Conclusão

Neste trabalho, foi apresentada uma discussão teórica e experimental sobre o monitoramento do envelhecimento de software relacionado à memória, focada especialmente em vazamentos de memória. Discutiu-se a respeito dos mecanismos fundamentais do monitoramento de vazamentos de memória por meio de experiências práticas. Foram mostrados achados recentes em termos de importantes desvantagens ao usar indicadores de envelhecimento conhecidos para monitorar efeitos de envelhecimento causados por vazamentos de memória. Foi possível apresentar duas novas abordagens práticas para superar tais limitações. Primeiro demonstrou-se que usar o total de memora livre/usada, bem como outros indicadores de sistema relacionados (ex. tamanho da área de *swap*) apresenta um elevado risco de realizar diagnósticos falso-positivos de envelhecimento. Se tal uso é um requisito importante, então é sugerida uma abordagem combinada, envolvendo indicadores de aplicação e de sistema (ex. memória de *buffer-cache*). Em relação a indicadores de envelhecimento específicos de aplicação, também foram demonstradas importantes limitações em se usar apenas o RSS, indicador de envelhecimento largamente utilizado atualmente. Portanto, propôs-se uma abordagem mais precisa para avaliar a existência de vazamento de memória, que consiste em analisar a combinação do tamanho virtual da aplicação (VIRT) e do tamanho do seu conjunto residente (RSS).

Ao executar esta abordagem, evidenciou-se que o comportamento do RSS pode levar ao falso diagnóstco de que não há envelhecimento de software. Através da análise, foi possível apontar que, mesmo quando o tamanho do conjunto residente do processo para de crescer, pode ser que o vazamento de memória continue a ocorrer, mas dentro da memória virtual da aplicação.

Referências

A. Macedo, T.F. Borges and, R. Matias. The mechanics of memory-related software aging. *IEEE Int'l Symp. on Software Reliability Eng*, 2010. Citado 6 vezes nas páginas 10, 12, 13, 15, 28 e 29.

B. Cantrill, M. Shapiro, and A. Leventhal. Dynamic instrumentation of production systems. *Proc. Annual Conference on USENIX Annual Technical Conference*, 2004. Citado na página 24.

B. Jacob, P. Larson, B. Leitão, and S.A.M.M Da Silva. Systemtap: instrumenting the linux kernel for analyzing performance and functional problems. *IBM Redbook*, 2008. Citado na página 24.

D. Controneo, R. Natella, and S. Russo. Software aging analysis of the linux operating system. *Proc. 21st IEEE International Symposium on Software Reliability*, p. 71–80, 2010. Citado na página 23.

F. Machida, A. Andrzejak, R. Matias and E. Vicente. On the effectiveness of mann-kendall test for detection of software aging. *International Workshop of Software Aging and Rejuvenation (ISSRE/WoSAR 2013), USA*. Citado na página 19.

Filebench. Citado na página 24.

G. Hunt and D. Brubacher. Detours: binary interception of win32 functions. *Proc. 3rd USENIX Windows NT Symposium, USENIX*, 1999. Citado na página 24.

J. Alonso, J.L. Berral, R. Gavaldà, and J. Torres. Adaptive on-line software aging prediction based on machine learning. *Proc. 40th Annual IEEE/IFIP Int'l Conference on Dependable Systems and Networks*, Mar. 2010. Citado na página 10.

J.C.T. Araujo, R. Matos, P.R.M. Maciel, R. Matias, and I. Beicker. Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure. *Proc. ACM/IFIP/USENIX 12th Int'l Middleware Conference, Industry Track*, v. 59, p. 102–114, 2011. Citado 3 vezes nas páginas 10, 12 e 29.

M. Grottke, R. Matias, and K. Trivedi. The fundamentals of software aging. *IEEE Int'l Symp. on Software Reliability Eng*, p. 1–6, 2008. Citado 3 vezes nas páginas 10, 20 e 22.

M.S. Johnstone and P.R. Wilson. The memory fragmentation problem: solved. *Proc. of Int'l Symp. on Memory Management*, 2008. Citado na página 10.

Q. Ni, W. Sun, and S. Ma. Memory leak detection in sun solaris os. *Proc. Int'l Symp. on Computer Science and Computational Technology*, 2008. Citado 2 vezes nas páginas 10 e 12.

R. Matias, A. Andrzejak, F. Machida, D. Elias and K Trivedi. A systematic approach for low-latency and robust detection of software aging. *33rd IEEE International Symposium on Reliable Distributed System*, p. 189–196, 2014. Citado na página 18.

R. Matias and P. J. Freitas Filho. An experimental study on software aging and rejuvenation in web servers. *Proc. 30th Annual Int'l Computer Software and Applications Conference*, v. 1, p. 189–196, 2006. Citado 6 vezes nas páginas 10, 12, 14, 22, 23 e 29.

R. Matias, I. Beicker, B. Leitão, and P.R.M. Maciel. Measuring software aging effects through os kernel instrumentation. *IEEE Int'l Symp. on Software Reliability Eng*, 2010. Citado 4 vezes nas páginas 12, 17, 23 e 29.

R. Matias, P. A. Barbetta, K. S. Trivedi and P. J. F. Filho. Accelerated degradation tests applied to software aging experiments. *IEEE Transaction on Reliability*, v. 59, p. 102–114, Mar. 2010. Citado 2 vezes nas páginas 10 e 29.

S. Garg, A. van Moorsel, K. S. Trivedi and K. S. Trivedi. A methodology for detection and estimation of software aging,. *Proc. 9th Int'l Symp. on Software Reliability Eng*, p. 283–292, 1998. Citado 2 vezes nas páginas 10 e 13.

T. Dohi, K. Goseva-Popstojanova, and K.S. Trivedi. Statistical non-parametric algorithms to estimate the optimal software rejuvenation schedule. *Proc. Pacific Rim Int'l Symposium on Dependable Computing*, 2000. Citado na página 22.

T.B. Ferreira, R. Matias, A. Macedo and L.B. Araujo. An experimental study on memory allocators in multicore and multithreaded applications. *Proc. 12th Int'l Conference on Parallel and Distributed Computing, Applications and Technologies*, 2011. Citado 5 vezes nas páginas 13, 25, 28, 29 e 30.

U. Vahalia. "unix internals: The new frontiers. *Prentice Hall*, 1995. Citado 2 vezes nas páginas 13 e 28.

W. Xie, Y. Hong, and K. S. Trivedi. Analysis of a two-level software rejuvenation policy. *Reliability Engineering and System Safety*, v. 87(1), p. 13–22, 2005. Citado na página 22.

Y. Bao, X. Sun, and K.S. Trivedi. A workload-based analysis of software aging, and rejuvenation. *IEEE Transactions on Reliability*, v. 54(3), p. 541–548, 2005. Citado na página 23.

Y. Huang, C. Kintala, N. Kolettis and N. Fulton. Software rejuvenation: analysis, module and applications. *Twenty-Fifth Int'l Symposium on Fault-Tolerant Computing*, p. 381–390, 1995. Citado na página 10.

Apêndices

APÊNDICE A – Algoritmo sem leak de memória do experimento 2

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

main(){
        int nr1, nr2, i, j;
        char *alocar;

        for(;;){
                nr1 = (rand() % 30) + 1;
                nr2 = (rand() % 5) + 1;
                printf("Alocando memória...\n");
                alocar = (char*)malloc(1024*nr2);
                if(alocar == NULL){
                        printf("Memória não alocada!!!!\n");
                }else{
                        for(j = 0; j < (1024*nr2); j++){
                                alocar[j] = 'A';
                        }
                }
                printf("Tempo de sleep agora: %d\n", nr1);
                sleep(nr1);
                printf("Desalocando memória...\n");
                free(alocar);
        }
}
```

APÊNDICE B – Algoritmo com leak de memória do experimento 2

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

main(){
        int nr1, nr2, nr3, i, j;
        char *alocar;

        for(;;){
                nr1 = (rand() % 30) + 1;
                nr2 = (rand() % 5) + 1;
                nr3 = (rand() % 100) + 1;
                printf("Alocando memória...\n");
                alocar = (char*)malloc(1024*nr2);

                if(alocar == NULL){
                        printf("Memória não alocada!!!!\n");
                }else{
                        for(j = 0; j< (1024*nr2); j++){
                                alocar[j] = 'A';
                        }
                }
                printf("Tempo de sleep agora: %d\n", nr1);
                sleep(nr1);
                if(nr3 % 2 == 0){
                        printf("%d par. Liberando memória...\n", nr3);
                        free(alocar);
                }
                else{
                        printf("%d impar. A memória não será desalocada...\n", nr3);
                }
        }
}
```

APÊNDICE C – Algoritmo com leak de memória do experimento 5

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

main(){
        int nr1, nr2, i, j;
        char *alocar;

        for(;;) {
                nr1 = (rand() % 15) + 1;
                nr2 = (rand() % 512) + 512;

                printf("Alocando memória...\n");
                alocar = (char*)malloc(1024*nr2);

                if(alocar == NULL) {
                        printf("Memória não alocada!!!!\n");
                } else {
                        for(j = 0; j< (1024*nr2); j++) {
                                alocar[j] = 'A';
                        }
                }
                printf("Tempo de sleep agora: %d\n", nr1);
                sleep(nr1);
        }
}
```