UNIVERSIDADE FEDERAL DO AMAZONAS - UFAM
DEPARTAMENTO DE ELETRÔNICA E TELECOMUNICAÇÕES
GRADUAÇÃO EM ENGENHARIA ELÉTRICA - TELECOMUNICAÇÕES

Investigando as Relações Entre Problemas de Segurança de Memória e Desempenho em Programas Kotlin

Daniel Yuri da Rocha Moura

Manaus - AM
Janeiro de 2023

Daniel Yuri da Rocha Moura

Investigando as Relações Entre Problemas de Segurança de Memória e Desempenho em Programas Kotlin

Projeto Final de Curso submetido à avaliação, como requisito parcial para a obtenção do título de Bacharel em Engenharia Elétrica - Telecomunicações da Universidade Federal do Amazonas.

Orientador

Lucas Carvalho Cordeiro, PhD

Universidade Federal do Amazonas - UFAM

Departamento de Eletrônica e Telecomunicações

Manaus - AM

Janeiro de 2023

Monografia de Graduação sob o título *Investigando as Relações Entre Problemas de Segurança de Memória e Desempenho em Programas Kotlin* apresentada por Daniel Yuri da Rocha Moura e aceita pelo Departamento de Eletrônica e Telecomunicações da Universidade Federal do Amazonas, sendo aprovada por todos os membros da banca examinadora abaixo especificada:

Lucas Carvalho Cordeiro, PhD

Orientador

Departamento de Eletrônica e Telecomunicações

Universidade Federal do Amazonas

Rosiane de Freitas Rodrigues, Dra.

Instituto de Computação

Universidade Federal do Amazonas

Francisco de Assis Pereira Januário, Dr.

Departamento de Eletrônica e Telecomunicações

Universidade Federal do Amazonas

Manaus - AM, 23 de Fevereiro de 2023.

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

M929i Moura, Daniel Yuri da Rocha

Investigando as relações entre problemas de segurança de memória e desempenho em programas kotlin / Daniel Yuri da Rocha Moura . 2023

62 f.: il. color; 31 cm.

Orientador: Lucas Carvalho Cordeiro

TCC de Graduação (Engenharia Elétrica - Telecomunicações) - Universidade Federal do Amazonas.

1. Kotlin. 2. Segurança de memória. 3. Desempenho de software. 4. Análise estática. 5. Análise dinâmica. I. Cordeiro, Lucas Carvalho. II. Universidade Federal do Amazonas III. Título

À minha família.

Agradecimentos

Agradeço primeiramente a Deus por nos dar essa oportunidade de aprendizado nessa vivência terrena. Agradeço à minha esposa Maria da Conceição, e aos meus filhos Jorge Gregório e Maria Clara, por serem minha fonte de afeto e alegria. Aos meus pais Donato e Maria Arlete pela criação e suporte incondicional, e ao meu irmão André Luiz, companheiro de sempre nas discussões sobre os caminhos da vida. Ao meu orientador, professor Lucas Cordeiro, e à professora Rosiane de Freitas, que sempre me impulsionaram para que eu pudesse dar o meu melhor nos estudos.

Peçam, e será dado; busquem, e encontrarão; batam, e a porta será aberta...

Mateus 7.

Investigando as Relações Entre Problemas de Segurança de Memória e Desempenho em Programas Kotlin

Autor: Daniel Yuri da Rocha Moura

Orientador: Lucas Carvalho Cordeiro, PhD

Resumo

O presente trabalho investiga de forma empírica as relações entre problemas de segurança de memória e desempenho em programas Kotlin. Para isso, foi feita uma pesquisa e descrição das principais fraquezas relacionadas à segurança de memória em sistemas de software. Em seguida, foi feito um levantamento e seleção de ferramentas de análise estática e dinâmica, que foram analisadas de acordo com suas capacidades para detecção de problemas de segurança de memória e análise de desempenho. Por último, são apresentados dois experimentos onde essas ferramentas são utilizadas para identificar problemas de segurança de memória em programas Kotlin e observar possíveis relações desses problemas com o desempenho desses programas. Os resultados mostram que a ferramenta ESBMC-Jimple pôde identificar estouros de pilha, estouros aritméticos e divisões por zero em programas Kotlin, mas em nenhum desses casos foi possível observar implicações no desempenho desses programas, pois a JVM interrompe a execução dos programas ao detectar esses problemas. Em outro experimento, foi possível observar problemas de desempenho em um programa Kotlin ocasionados por um problema de segurança de memória, mas só foi possível identificar tal problema através de revisão manual de código, pois a ferramenta ESBMC-Jimple não possui suporte para a identificação do relativo problema.

Palavras-chave: Kotlin, Segurança de Memória, Desempenho de Software, Análise Estática, Análise Dinâmica.

Investigando as Relações Entre Problemas de Segurança de Memória e Desempenho em Programas Kotlin

Autor: Daniel Yuri da Rocha Moura

Orientador: Lucas Carvalho Cordeiro, PhD

Abstract

The present work empirically investigates the relationship between memory safety issues and performance in Kotlin programs. For this, a research and description of the main weaknesses related to memory safety in software systems was carried out. Then, a survey and selection of static and dynamic analysis tools were carried out, which were analyzed according to their capabilities for detecting memory safety problems and performance analysis. Finally, two experiments are presented where these tools are used to identify memory safety problems in Kotlin programs and to observe possible relationships between these problems and the performance of these programs. The results show that the ESBMC-Jimple tool was able to identify stack overflows, arithmetic overflows and division by zero in Kotlin programs, but in none of these cases was it possible to observe implications for the performance of these programs, since the JVM interrupts the execution of the programs when detecting these problems. In another experiment, it was possible to observe performance problems in a Kotlin program caused by a memory safety problem, but it was only possible to identify such problem through manual code review, as the ESBMC-Jimple tool does not have support for identifying the relative problem.

Keywords: Kotlin, Memory Safety, Software Performance, Static Analysis, Dynamic Analysis.

Lista de ilustrações

Figura 1 – Diagrama da metodologia utilizada no trabalho 18
Figura 2 – Diagrama de blocos mostrando a visão geral da arquitetura do ESBMC-Jimple. 34
Figura 3 – Captura de tela do Intellij Idea com aba do Inspector reportando problemas no código. 34
Figura 4 – Captura de tela mostrando o JProfiler em atividade. 35
Figura 5 – Captura de tela mostrando o Android Profiler em atividade. 36
Figura 6 – Captura de tela do monitoramento de memória do JProfiler. 40
Figura 7 – Captura de tela do monitoramento de memória no Android Profiler. 41
Figura 8 – Captura de tela do monitoramento de CPU do JProfiler. 41
Figura 9 – Captura de tela do monitoramento de uso de CPU no Android Profiler. 42
Figura 10 – Captura de tela mostrando a atividade do coletor de lixo no JProfiler. 42
Figura 11 – Captura de tela mostrando a atividade do coletor de lixo no Android Profiler. 43
Figura 12 – Call tracer do JProfiler mostrando as múltiplas chamadas da função recursiva. 45
Figura 13 – Trace de memória mostrando o consumo crescente de memória heap. 47
Figura 14 – Tela mostrando o trace de memória e a atuação do coletor de lixo. . . 48
Figura 15 – Tela mostrando o coletor de lixo em frequente atividade. 48
Figura 16 – Número de instâncias e tamanho ocupado pela Classe Objeto (em destaque). 48
Figura 17 – Momento em que o limite de memória é atingido. 49

Figura 18 – Consumo de memória no programa *ListaDeObjetosLimitad*a. 49
Figura 19 – Momento em que o coletor de lixo atua no programa *ListaDeObjetos-Limitad*a. 50

Lista de tabelas

Tabela 1 – Vulnerabilidades comuns relacionadas à memória categorizadas pelo sistema CWE. 31
Tabela 2 – Resultados da análise comparativa entre as ferramentas de detecção de problemas de segurança de memória 39
Tabela 3 – Características gerais do JProfiler e do Android Profiler. 40
Tabela 4 – Tabela mostrando o número de chamadas recursivas da função *recursiveFunction()* em cada espaço de tempo de captura. 46

SUMÁRIO

1 INTRODUÇÃO . . . 15

1.1 Definição do problema . . . 17

1.2 Objetivos . . . 17

1.3 Metodologia . . . 18

1.4 Organização do Trabalho . . . 19

2 FUNDAMENTAÇÃO TEÓRICA . . . 21

2.1 Linguagem Kotlin . . . 21

2.1.1 Sintaxe e estruturas . . . 22

2.2 Segurança de Memória . . . 23

2.2.1 Segurança de Memória na Máquina Virtual Java (JVM) . . . 24

2.3 Desempenho de Software . . . 24

2.4 Análise de Software . . . 25

2.4.1 Análise Estática . . . 25

2.4.2 Análise Dinâmica . . . 26

3 TRABALHOS RELACIONADOS . . . 27

3.1 Trabalhos relacionados ao uso e avaliação de ferramentas de detecção de problemas de segurança de memória . . . 27

3.2 Trabalhos relacionados ao uso e avaliação de ferramentas de análise de desempenho . . . 28

4 INVESTIGANDO AS RELAÇÕES ENTRE PROBLEMAS DE SEGURANÇA DE MEMÓRIA E DESEMPENHO EM PROGRAMAS KOTLIN . . . 30

4.1 Identificação e Descrição de Problemas Comuns Relacionados à Segurança de Memória . . . 30
4.1.1 Estouro de Inteiros (*Integer Overflow*) . . . 31
4.1.2 Divisão por zero . . . 32
4.1.3 Violação dos limites de um array . . . 32
4.1.4 Vazamento de memória (*Memory Leak*) . . . 32
4.1.5 Estouro de pilha . . . 32
4.2 Seleção das Ferramentas . . . 33
4.2.1 Descrição das Ferramentas Selecionadas . . . 33
4.2.1.1 ESBMC-Jimple . . . 33
4.2.1.2 Intellij Idea Inspector . . . 34
4.2.1.3 JProfiler . . . 34
4.2.1.4 Android Profiler . . . 35
4.3 Avaliação Experimental . . . 36
4.3.1 Configurações Gerais dos Experimentos . . . 36
4.3.2 Descrição das *benchmarks* . . . 37
4.3.3 Análise das Ferramentas de Detecção de Problemas de Segurança de Memória . . . 37
4.3.4 Análise das Ferramentas de Monitoramento de Desempenho . . . 39
4.3.5 Estudo Empírico Investigando Relações Entre Problemas de Segurança de Memória e Desempenho em Programas Kotlin . . . 44
4.3.5.1 Experimento 1: Analisando um programa com estouro de pilha . . . 44
4.3.5.2 Resultados e Discussão do Experimento 1 . . . 45
4.3.5.3 Experimento 2: programa com consumo crescente e ininterrupto de memória 46
4.3.5.4 Resultados e Discussão do Experimento 2 . . . 47
4.4 Ameaças à validade . . . 50

5 CONCLUSÕES . . . 52
5.1 Trabalhos Futuros . . . 53

Referências . . . 54

APÊNDICE A BENCHMARKS 59

1

INTRODUÇÃO

Kotlin (FOUNDATION, 2022) é uma linguagem de programação moderna cuja popularidade está crescendo. A Google a considerou como a segunda linguagem oficial para o desenvolvimento de aplicativos Android (OLIVEIRA; TEIXEIRA; EBERT, 2020), um dos sistemas operacionais mais utilizados em dispositivos de telefonia móvel, despertando o interesse de empresas, desenvolvedores e do público geral desde o seu lançamento oficial (GANDHEWAR; SHEIKH, 2011). Por ser uma linguagem que executa na Máquina Virtual Java (JVM), ela herda o gerenciamento automático de memória da linguagem Java, utilizando um "coletor de lixo" (Garbage Collector) para desalocar objetos que não estão mais sendo referenciados (Oracle, 2023), evitando o esgotamento dos recursos de memória que podem levar a problemas de desempenho. Ela também possui uma verificação de segurança de nulos, um recurso de segurança de memória que elimina o perigo de referências nulas (NullPointerException) (KHAN; KUCHERENKO, 2018) (FOUNDATION, 2022). Quando falamos em segurança de memória em linguagens de programação, nos referimos à capacidade de utilizar os recursos de memória de forma eficiente, de forma a não deixar vulnerabilidades e evitando que falhas possam vir a ocorrer. Invasores podem afetar esses pilares através da exploração dessas vulnerabilidades, podendo ter acesso a dados sigilosos (SILVA et al., 2013a; PEREIRA et al., 2016; MONTEIRO et al., 2017). Muitos problemas de segurança de memória estão frequentemente relacionados aos aspectos de confiabilidade e segurança do sistema, que são aspectos da qualidade de software (ROCHA et al., 2020; MONTEIRO; GADELHA; CORDEIRO, 2022; CORDEIRO, 2021). Em alguns casos, podem estar diretamente

relacionados a problemas de desempenho, como no caso de um erro de memory leak, que pode causar o esgotamento dos recursos de memória e expor vulnerabilidades no sistema, permitindo, por exemplo, que um usuário possa se aproveitar de um comportamento inesperado do programa para fins maliciosos . Além disso, alguns sistemas possuem técnicas para evitar a exploração de vulnerabilidades, podendo causar alguma sobrecarga no sistema (UNIDOS, 2022) (MARTINS, 2009) e afetando o desempenho. Para garantir a qualidade de um sistema de software, existem técnicas manuais e automáticas de análise de software (GADELHA; MENEZES; CORDEIRO, 2021; CORDEIRO; FILHO; BESSA, 2020). Técnicas manuais, como a inspeção de código, podem ser muito úteis no processo de identificação de vulnerabilidades, mas demandam tempo, paciência e habilidade dos desenvolvedores. Ferramentas automáticas de análises estática e dinâmica de programas auxiliam o programador no desenvolvimento de programas de qualidade ao longo de todas as etapas do desenvolvimento e são bem mais rápidas que as técnicas manuais. Essas ferramentas podem realizar monitoramento de recursos de memória e uso de CPU, identificação de problemas de segurança de memória, entre outras funcionalidades. Desenvolvedores da linguagem Kotlin dispõem de algumas ferramentas para análise estática e dinâmica, como o JProfiler (ej-technologies GmbH, 2023), o Android Profiler (Google, 2023), o ESBMC-Jimple (MENEZES et al., 2022) e o Intellij Idea Inspector (JetBrains, 2023). A área científica ainda carece de estudos a respeito dessas ferramentas, sobre sua efetividade na identificação de problemas de segurança de memória e desempenho e, além disso, se podemos observar relações entre esses problemas através do uso combinado dessas ferramentas. O presente trabalho traz como principais contribuições uma análise de ferramentas de análises estática e dinâmica para programas Kotlin e ao final apresenta dois experimentos com o intuito de utilizar essas ferramentas para analisar problemas de memória e possíveis relações com problemas de desempenho em programas Kotlin . O trabalho foi desenvolvido dentro do projeto de pesquisa, desenvolvimento e inovação intitulado Técnicas de Inteligência Artificial Para Análise e Otimização de Desempenho de Software - "SW-Perfi", realizado em parceria entre a UFAM, a Motorola e a Flextronics.

1.1 Definição do problema

Problemas de segurança de memória são conhecidos por deixarem vulnerabilidades que podem ser exploradas por usuários maliciosos com o intuito de obtenção de dados sigilosos ou abalar a integridade dos sistemas. No entanto, alguns desses problemas de segurança de memória podem ocasionar problemas de desempenho em sistemas de software sem que haja a necessidade de exploração da vulnerabilidade (WHITTAKER; THOMPSON, 2003.), como esgotamento de recursos e lentidão na execução.

Para garantir a qualidade e evitar problemas de segurança de memória e desempenho nos programas, foram desenvolvidas diversas ferramentas de análises estática e dinâmica capazes de identificar e reportar essas questões. No entanto, existem poucos estudos analisando tais ferramentas com suporte para a linguagem Kotlin e se podem ser usadas de forma complementar para observar com profundidade problemas de segurança de memória e as possíveis implicações desses problemas no desempenho dos programas.

1.2 Objetivos

O objetivo geral do presente trabalho é investigar, através de um estudo empírico, problemas de segurança de memória em programas Kotlin e se há a possibilidade de observar possíveis relações diretas com problemas de desempenho nesses programas através do uso de ferramentas de análise estática e dinâmica.

Os objetivos específicos são:

a) Identificar e descrever alguns dos problemas de segurança de memória mais graves e recorrentes e suas possíveis consequências nos sistemas de software;

b) Analisar ferramentas de análise estática e dinâmica que são capazes de apresentar dados relacionados a problemas de segurança de memória e desempenho em programas Kotlin;

c) Apresentar resultados referentes a experimentos realizados buscando combinar as funcionalidades das ferramentas analisadas, com o objetivo de identificar

problemas de segurança de memória, analisar o desempenho desses programas e verificar se há relações entre esses problemas.

1.3 Metodologia

A metodologia proposta no presente trabalho utiliza aspectos teóricos e práticos para chegar a um entendimento sobre desempenho e segurança de memória em programas Kotlin e como essas questões podem estar relacionadas. A Figura 1 mostra um diagrama de blocos com uma visão geral da metodologia proposta:

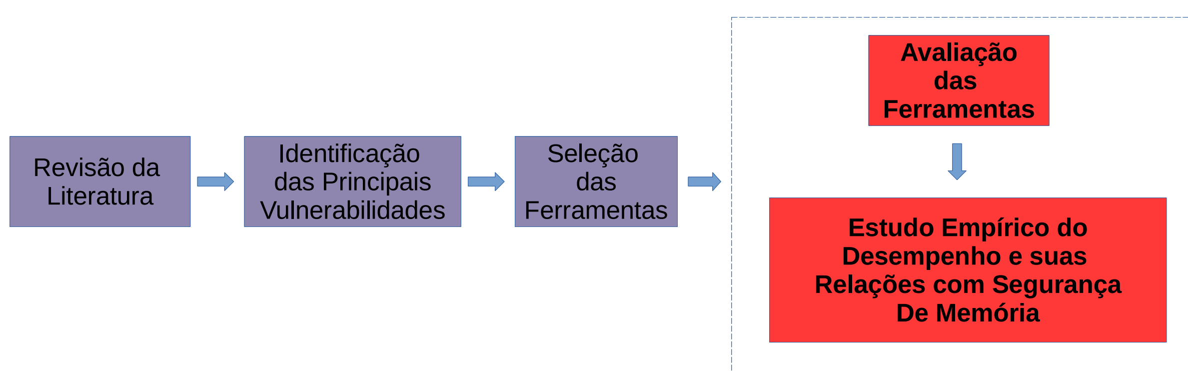

Figura 1 – Diagrama da metodologia utilizada no trabalho

O diagrama de blocos da metodologia utilizada no trabalho é descrito da seguinte forma:

a) Revisão da literatura: Realização de pesquisas em livros e trabalhos científicos com o intuito de obter embasamento teórico para o desenvolvimento do presente estudo.

b) Descrição de problemas de segurança de memória comuns: Busca em trabalhos científicos e em sites e fóruns especializados da Internet com o intuito de identificar e descrever algumas das vulnerabilidades mais comuns e significativas relacionadas à memória em sistemas de software. Essa etapa servirá como base para a seleção e desenvolvimento dos casos de teste que serão usados na parte experimental da pesquisa.

c) Seleção das ferramentas: Identificação e seleção de ferramentas de análises estática e dinâmica de programas Kotlin. O critério de seleção é norteado por duas questões, que são respondidas com base na análise da documentação da ferramenta,

conteúdo de fóruns de discussão na Internet e entrevista informal com desenvolvedores das ferramentas.

d) Análise das ferramentas: Análise de ferramentas de análise estática e dinâmica, com o objetivo de obter informações a respeito de sua eficácia na detecção de problemas de segurança de memória e no monitoramento de dados relativos à métricas de desempenho em programas Kotlin.

e) Estudo empírico de segurança de memória e suas relações com questões de desempenho: Experimentos utilizando as ferramentas de análise estática com o intuito de investigar possíveis implicações de problemas de segurança de memória no desempenho de programas Kotlin.

1.4 Organização do Trabalho

Esta seção descreve a forma como o trabalho está organizado, começando pelo presente capítulo, que contém a Introdução, a Definição do Problema e os Objetivos Gerais e Específicos do trabalho e a metodologia utilizada.

No Capítulo 2 são apresentados alguns conceitos fundamentais para o trabalho, começando por uma descrição da linguagem Kotlin, com sua sintaxe e estruturas. Discorre também sobre a questão da segurança de memória, com alguns aspectos gerais a respeito e aspectos específicos de segurança de memória na Máquina Virtual Java (JVM). Na sequência, expõe alguns conceitos a respeito de desempenho de software. Por último, aborda conceitos de análise de software, discorrendo a respeito de técnicas de análise estática e dinâmica.

O Capítulo 3 apresenta alguns trabalhos relacionados, considerando-se os trabalhos que analisam e exploram o uso de ferramentas de análise estática e dinâmica com ênfase em detecção de problemas de segurança de memória e desempenho em sistemas de software.

O Capítulo 4 começa descrevendo como se procedeu a análise das ferramentas utilizadas no estudo e como foram utilizadas para investigação de possíveis relações entre problemas de segurança de memória e desempenho em programas Kotlin . Na

sequência, apresenta um estudo que busca identificar e descrever algumas das vulnerabilidades de memória mais perigosas e comuns em sistemas de software e suas consequências. O Capítulo também possui uma seção com a seleção das ferramentas utilizadas na parte experimental e uma descrição dessas ferramentas. Ao final do Capítulo, é apresentada a avaliação experimental, com os resultados e discussão, além de uma seção de ameaças à validade.

Por fim, no Capítulo 5 são expostas as conclusões e sugestões para trabalhos futuros.

2

Fundamentação Teórica

Este capítulo contém alguns conceitos básicos sobre a linguagem Kotlin, com uma descrição da sua sintaxe e estruturas. Na sequência há uma seção descrevendo o problema de segurança de memória, abordando primeiramente um contexto geral e depois aspectos específicos de segurança de memória da Máquina Virtual Java. Em seguida, são apresentados conceitos a respeito de desempenho de software. Por último, há uma descrição de técnicas de análise de software, com conceitos de análises estática e dinâmica.

2.1 Linguagem Kotlin

Kotlin (FOUNDATION, 2022) é uma linguagem de programação moderna, concisa e orientada a objetos, desenvolvida pela JetBrains em 2011, com o objetivo de melhorar a produtividade dos desenvolvedores e eliminar algumas das deficiências de outras linguagens existentes (FOUNDATION, 2022), como verbosidade e problemas com ponteiros nulos. A linguagem Kotlin é de tipagem estática, suporta inferência de tipos, é compatível com Java e é executada em uma JVM (Java Virtual Machine), permitindo que os programas Kotlin sejam executados em qualquer lugar onde a JVM estiver disponível (FOUNDATION, 2022) (KHAN; KUCHERENKO, 2018).

A interoperabilidade com Java é uma das vantagens do Kotlin. A linguagem foi projetada para ser totalmente compatível com o código Java existente (FOUNDATION, 2022), permitindo que os desenvolvedores usem bibliotecas e frameworks Java em

seus projetos Kotlin sem precisar fazer grandes mudanças. Além disso, o Kotlin oferece recursos que simplificam o código Java, como funções de extensão, null safety e lambdas (FOUNDATION, 2022) (KHAN; KUCHERENKO, 2018).

2.1.1 Sintaxe e estruturas

Uma das características mais marcantes do Kotlin é a sua simplicidade (FOUNDATION, 2022). A linguagem tem uma sintaxe limpa e concisa, o que facilita a leitura e a escrita de código. Por exemplo, a declaração de variáveis em Kotlin é feita usando a palavra-chave "var"ou "val", seguida do nome da variável e do valor inicial, como em "var nome = "Kotlin. Além disso, a sintaxe de funções em Kotlin é bastante clara e simples, com a palavra-chave "fun", seguida do nome da função, parâmetros entre parênteses e um corpo de função dentro de chaves. Um exemplo dessa sintaxe pode ser visto no Algoritmo 2.1.

Algoritmo 2.1 – Exemplo de sintaxe da linguagem Kotlin.

```
1 class Pessoa(val nome: String, var idade: Int) {
2     fun cumprimentar() {
3         println("Oi, eu me chamo $nome e tenho $idade
              anos.")
4     }
5 }
```

Outra característica importante da sintaxe do Kotlin é a capacidade de evitar os perigosos *null pointers* (em português, "ponteiros nulos"), que são comuns em outras linguagens de programação. O Kotlin possui um sistema de tipos forte e rigoroso, que permite que os desenvolvedores especifiquem se uma variável pode ou não ser nula. Isso é feito adicionando um ponto de interrogação após o tipo da variável, como em "var nome: String? = null". Com essa especificação, o compilador do Kotlin irá apontar um erro caso haja tentativas de acessar variáveis nulas (FOUNDATION, 2022) (KHAN; KUCHERENKO, 2018).

O Kotlin também possui uma série de recursos avançados, como as funções

de extensão, que permitem adicionar novas funcionalidades a uma classe existente sem precisar alterar o código original. Além disso, a linguagem oferece suporte para programação assíncrona, por meio de recursos como corrotinas (FOUNDATION, 2022), que tornam mais fácil lidar com tarefas que exigem processamento paralelo.

2.2 Segurança de Memória

A segurança de software é um assunto cada vez mais relevante (JONES, 2007), dado ao crescente número de sistemas de software. Está relacionada a como o programa gerencia recursos de memória (UNIDOS, 2022) . Os problemas de segurança de memória estão entre as principais preocupações da segurança de software e podem ter consequências graves, incluindo vazamento de dados confidenciais, baixo desempenho, e até mesmo a possibilidade de execução de código malicioso. Tais problemas podem custar muito dinheiro, como no caso do foguete Ariane-5, onde uma falha de software causou sua explosão (SILVA et al., 2013b), custando milhões de dólares. Podem também custar vidas, como no caso de falhas de software em dispositivos médicos.

Existem vários tipos de problemas de segurança de memória, incluindo estouro de *buffer*, estouro aritmético, vazamento de memória, acesso inválido à memória, entre outros (DHURJATI et al., 2003). Por exemplo, o estouro de *buffer* ocorre quando um programa tenta escrever mais dados em um buffer do que ele pode armazenar, o que pode levar à corrupção de dados ou até mesmo a falhas de segurança. Já o vazamento de memória ocorre quando um programa aloca memória, mas não a libera quando não é mais necessária (TANG; PLSEK; VITEK, 2012), o que pode levar a uma redução de desempenho ao longo do tempo ou até mesmo a uma falha completa do sistema. Entre as possibilidades para a solução desses problemas está o uso de técnicas e ferramentas de análise estática e dinâmica, que podem detectar e corrigir problemas de segurança de memória antes que se tornem vulnerabilidades que possam ser exploradas.

2.2.1 Segurança de Memória na Máquina Virtual Java (JVM)

A JVM é a base da plataforma Java (SINGH, 2014), e é responsável por gerenciar a alocação de memória e as operações de acesso à memória. Como a memória é uma parte fundamental do funcionamento de qualquer aplicativo, é essencial garantir que a segurança da memória seja uma prioridade durante o desenvolvimento.

A JVM utiliza um modelo de segurança baseado em *sandboxes*, que são ambientes isolados que limitam o acesso do código Java a recursos do sistema. Esses sandboxes garantem que o código Java não possa acessar áreas de memória que não foram alocadas para ele, impedindo que um aplicativo malicioso corrompa a memória ou acesse dados sensíveis (Oracle, 2023) (COFFIN, 2011).

Além disso, a JVM possui um mecanismo de gerenciamento automático de memória que utiliza coleta de lixo para liberar automaticamente a memória que não está mais sendo usada pelo aplicativo (Oracle, 2023). Isso ajuda a prevenir vazamentos de memória e evita que um aplicativo utilize mais memória do que deveria, o que poderia causar problemas de desempenho e até mesmo travamentos. Em algumas outras linguagens, como C e C++, esse gerenciamento de memória é de responsabilidade do desenvolvedor (Oracle, 2023)

A JVM também possui um sistema de tratamento de exceções como medida de segurança. Por exemplo, de acordo com as especificações, a JVM pode ter um tamanho de pilha fixa ou também pode ser expandida. Caso o limite da pilha seja atingido, uma exceção do tipo *StackOverflowError* é lançada (Oracle, 2023), terminando a execução do programa.

2.3 Desempenho de Software

O desempenho é um dos mais importantes aspectos da qualidade de software. O desempenho afeta diretamente a experiência do usuário (RASHID; MAHMOOD; NISAR, 2019) e pode ter um impacto significativo em empresas de desenvolvimento de software. É por isso que o desempenho de software é uma preocupação importante para as empresas de software em todo o mundo, especialmente quando se trabalha

com programas que manipulam variáveis de ponto-flutuante (GADELHA; CORDEIRO; NICOLE, 2020; GADELHA et al., 2020). Cada avaliação de desempenho requer um conhecimento detalhado do sistema, além de uma escolha cuidadosa de metodologias, cargas de trabalho e ferramentas de análise (LEAL, 2015).

O desempenho do software pode ser medido de várias maneiras, incluindo tempo de resposta, taxa de transferência, utilização de recursos e escalabilidade. O tempo de resposta é a medida do tempo que leva para tarefas específicas serem concluídas em um aplicativo ou sistema. A utilização de recursos refere-se ao uso de recursos do sistema, como CPU, memória e armazenamento. A escalabilidade é a capacidade de um sistema lidar com uma carga de trabalho crescente sem perder desempenho.

Existem várias técnicas que os desenvolvedores podem usar para melhorar o desempenho do software. Uma das técnicas mais comuns é a otimização do código (PATIL et al., 2022), que envolve a análise do código para identificar áreas que podem ser melhoradas para reduzir o tempo de execução e melhorar a eficiência. Isso pode incluir a remoção de operações desnecessárias, a otimização de algoritmos e a melhoria da gestão de memória.

Os desenvolvedores podem usar ferramentas de monitoramento e análise de desempenho para identificar áreas problemáticas e ajustar o desempenho do software. Essas ferramentas podem monitorar a utilização de recursos do sistema, tempo de resposta, taxa de transferência e outros indicadores de desempenho para ajudar os desenvolvedores a identificar gargalos de desempenho e possíveis áreas para otimização (SEMICONDUCTORS, 2011).

2.4 Análise de Software

2.4.1 Análise Estática

A análise estática é uma técnica usada para avaliar o software sem executá-lo. É uma ferramenta poderosa para identificar possíveis bugs, vulnerabilidades de segurança e problemas de desempenho em software. Uma das principais vantagens da análise estática é que ela não causa uma sobrecarga direta (*overhead*) em tempo

de execução (SILVA et al., 2013b) e pode identificar uma ampla gama de potenciais problemas, incluindo *memory leaks*, que ocorrem quando um objeto que não está mais sendo referenciado continua ocupando espaço na memória de forma desnecessária, estouros aritméticos, *buffer overflows*, que ocorrem quando o limite de um buffer é excedido e há sobrescrição de memória adjacente, e *null pointer exceptions*, que ocorrem quando um programa tenta acessar um objeto de memória que não foi inicializado. Por outro lado, a análise estática não dispõe de informações geradas em tempo de execução (SILVA et al., 2013b). A análise estática pode ser usado em conjunto com outras técnicas, como análise dinâmica e revisão manual do código.

Entre as técnicas de análise estática estão a verificação de tipos, propagação de variáveis, análise de fluxo de dados (BODDEN, 2012) e a verificação formal(BAIER; KATOEN, 2008) (SILVA et al., 2021). Alguns exemplos de ferramentas de análise estática são o ESBMC (CORDEIRO et al., 2012; MONTEIRO; GADELHA; CORDEIRO, 2022), JBMC (CORDEIRO et al., 2018; CORDEIRO; KROENING; SCHRAMMEL, 2019), o Jayhorn (KAHSAI et al., 2016), o Intellij Idea Inspector (JetBrains, 2023) e o Detekt (ARTIFACTS, 2021).

2.4.2 Análise Dinâmica

A análise dinâmica é uma técnica usada avaliar o software analisando seu comportamento enquanto ele está em execução, sendo essa uma das principais vantagens, pois pode tirar proveito de informações só disponíveis em tempo de execução (SILVA et al., 2013b). É uma ferramenta poderosa para identificar gargalos de desempenho, *memory leaks* e outros problemas que podem não ser visíveis através de análise estática, que são em geral problemas que ocorrem em tempo de execução. Entre as ferramentas de análise dinâmica mais completas estão os *profilers*, que utilizam técnicas como a instrumentação e a amostragem (SEMICONDUCTORS, 2011) para monitorar a utilização de recursos de memória, utilização de CPU e energia consumida pela aplicação. Entre os principais profilers estão o JProfiler (ej-technologies GmbH, 2023), o Android Profiler (Google, 2023) e o Yourkit (YOURKIT, 2021).

3

TRABALHOS RELACIONADOS

O presente Capítulo traz um resumo de trabalhos relacionados com o presente estudo, alguns deles abordando o uso e avaliação de ferramentas de análises estática e dinâmica com ênfase em detecção de problemas de segurança de memória e outros analisando o uso e comparando ferramentas que possuem ênfase em análise de desempenho.

3.1 Trabalhos relacionados ao uso e avaliação de ferramentas de detecção de problemas de segurança de memória

Em "*Evaluation of Static Analysis Tools for Software Security*", os autores (ALBREIKI; MAHMOUD, 2014) definem pontos fracos comuns de software com base no CWE/-SANS Top 25, OWASP Top Ten e pontos fracos do código-fonte NIST para avaliar as ferramentas de análise de segurança usando a metodologia NIST Software Assurance Metrics and Tool Evaluation (SAMATE). Os resultados mostram que as ferramentas de análise estática são, até certo ponto, eficazes em detecção de falhas de segurança no código-fonte; analisadores de código-fonte são capazes de detectar mais pontos fracos do que analisadores de *bytecode* e código binário; e enquanto as ferramentas podem auxiliar a equipe de desenvolvimento em atividades de revisão de aspectos relacionados à segurança do código, elas não são suficientes para descobrir todas as

fraquezas comuns em software.

Em "*Evaluating and comparing memory error vulnerability detectors*" (NONG et al., 2021) comparam cinco ferramentas de análises estática e dinâmica com capacidade para detecção de erros de memória utilizando dois conjuntos de benchmarks em linguagem C/C++. Resultados mostram que, embora geralmente rápidos, esses detectores tinham precisão bastante variada em diferentes categorias de vulnerabilidade e precisão geral moderada. Código complexo (por exemplo, *loops* profundos e recursões) e estruturas de dados (por exemplo, listas vinculadas profundamente incorporadas) pareciam ser barreiras comuns e importantes. A análise híbrida nem sempre superou a análise puramente estática ou dinâmica para detecção de vulnerabilidades de memória. No entanto, os resultados da avaliação foram visivelmente diferentes entre os dois conjuntos de dados usados. No caso, os estudos explicaram ainda mais as variações de desempenho entre esses detectores e permitiram recomendações para melhorias

Em "*Evaluation of Static Analysis Tools for Finding Vulnerabilities in Java and C/C++ Source Code*" (MAHMOOD; MAHMOUD, 2018), os autores avaliam ferramentas de análise estática com suporte para as linguagens C/C++ e Java sob os critérios de facilidade de uso, efetividade e suporte, mostrando os prós e contras de cada uma.

3.2 Trabalhos relacionados ao uso e avaliação de ferramentas de análise de desempenho

Em "*Evaluating the Accuracy of Java Profilers*", os autores (MYTKOWICZ et al., 2010) mostram que ferramentas comumente usadas para profiling de programas Java (xprof , hprof , JProfiler, and Yourkit) discordem entre si em suas medidas. O estudo propõe um metodologia para avaliar tais profilers sob o critério "acionável", que seria um termo mais adequado que "correto", devido à dificuldade de definir a corretude de uma medida. O processo é feito através de análise de causalidade. Os resultados mostraram que um *profiler* que coleta amostras aleatoriamente possui resultados mais acionáveis. Mostraram também que, usando vários estudos de caso, o *profiler* identifica corretamente os métodos que são importantes para otimizar; em alguns casos, outros

profilers relatam que esses métodos são frios e, portanto, não vale a pena otimizar.

Em "*Evaluation of Java Profiler Tools*", os autores (FLAIG; HERTL; KRÜGER, 2013) comparam os *profilers* JVM Monitor, Visual VM, Netbeans Profiler, JProfiler, Eclipse TPTP, Yourkit, Memory Analyzer e Mission Control, todos com suporte para a linguagem Java. Todos os profilers em teste foram inspecionados com um conjunto semelhante de testes para obter resultados a fim de criar comparações relativas. Foi criado relatório detalhado com os resultados do estudo para obter uma visão mais profunda sobre o comportamento funcional de cada *profiler*. Os resultados mostraram que três *profilers* se destacaram por suas convincentes capacidades. Assim, tornou-se difícil fazer uma recomendação para um único *profiler*, sendo preciso avaliar o contexto em que a ferramenta será usada. Segundo os autores, um *profiler* perfeito precisaria oferecer o melhor solução para qualquer cenário imaginável, sendo este um feito impossível de desenvolver.

4

Investigando as relações entre problemas de segurança de memória e desempenho em programas Kotlin

Este Capítulo apresenta o desenvolvimento de uma metodologia para investigação das relações entre problemas de segurança de memória e desempenho em programas Kotlin, que se inicia com uma identificação e descrição de alguns dos principais problemas relacionados à segurança de memória. Em seguida, há uma etapa de seleção de ferramentas, que passarão por uma análise comparativa. Tal análise comparativa fornecerá informações para a realização de um estudo empírico combinando essas ferramentas com o objetivo de observar possíveis implicações de problemas de segurança de memória no desempenho de programas Kotlin.

4.1 Identificação e Descrição de Problemas Comuns Relacionados à Segurança de Memória

Esta seção traz uma breve descrição de problemas comuns relacionados à memória e comumente presentes em avaliações de ferramentas de análise estática e dinâmica

(NONG et al., 2021) (MAHMOOD; MAHMOUD, 2018) (MENEZES et al., 2022) e mostra a identificação de cada uma de acordo com o sistema de categorização CWE (*Common Weakness Enumeration*) (MITRE Corporation, 2023). Tais problemas de memória serão considerados para a seleção dos casos de teste da etapa de avaliação experimental.

Tabela 1 – Vulnerabilidades comuns relacionadas à memória categorizadas pelo sistema CWE.

CWE-ID	**Name**
CWE-190	Integer Overflow or Wraparound
CWE-369	Divide by Zero
CWE-129	Improper Validation of Array Index
CWE-401	Missing Release of Memory After Effective Lifetime
CWE-121	Stack-based Buffer Overflow

Fonte: Própria.

Algumas das vulnerabilidades apresentadas na Tabela 1 já estiveram presentes na classificação das vulnerabilidades mais graves CWE Top 25, feito pela Homeland Security Systems Engineering, como no caso da CWE-190, que entrou na lista do ano de 2021.

4.1.1 Estouro de Inteiros (*Integer Overflow*)

O estouro de inteiros é um tipo de estouro aritmético que se caracteriza pela atribuição de um valor inteiro que não pode ser representado corretamente pelo tipo de dado ao qual está sendo atribuído, gerando erros de cálculo que podem resultar em vulnerabilidades, como no caso onde um desses valores é usado para calcular o tamanho de um buffer, podendo levar a um estouro de *buffer*. Em um caso famoso, um estouro de inteiros foi a causa da explosão do foguete Ariane 5, após um erro de precisão nos cálculos (SILVA et al., 2013b). Sua identificação no CWE corresponde ao CWE-190.

4.1.2 Divisão por zero

Acontece quando se tenta dividir um valor por zero. Tal situação pode levar a resultados inesperados, como consumo excessivo de recursos e ao término do programa, como no caso do navio da marinha americana USS Yorktown em 1997 (GCN, 1998), onde uma divisão por zero levou a um estouro de *buffer*, causando falhas no sistema de propulsão do navio. Sua identificação no CWE corresponde ao CWE-369.

4.1.3 Violação dos limites de um array

Ocorre quando o programa tenta acessar um índice inválido do array. A falta de checagem dos limites de um array pelo compilador podem levar a um estouro de buffer (LAM; CHIUEH, 2005), expondo vulnerabilidades. Sua identificação no CWE corresponde ao CWE-129.

4.1.4 Vazamento de memória (*Memory Leak*)

Ocorre quando objetos alocados dinamicamente na memória não são desalocados depois de seu uso (GANGWAR; KATAL, 2021), ocupando espaço na memória desnecessariamente. Tal problema pode levar a um excessivo consumo de memória, gerando problemas de desempenho. Também pode expor vulnerabilidades, que podem ser exploradas por usuários maliciosos. Sua identificação no CWE corresponde ao CWE-401.

4.1.5 Estouro de pilha

Ocorre quando o tamanho reservado para a pilha não é suficiente, gerando um comportamento indefinido no programa. Tal problema pode ocasionar corrupção de dados e expor vulnerabilidades no sistema, que podem ser exploradas por usuários maliciosos. É um dos tipos de vulnerabilidade mais explorados do mundo (MARTINS, 2009). Sua identificação no CWE corresponde ao CWE-121.

4.2 Seleção das Ferramentas

Para o presente estudo, foram selecionadas ferramentas com capacidade para identificação de problemas de segurança de memória em programas Kotlin. Para a seleção das ferramentas com ênfase em monitoramento de desempenho, optou-se por considerar os profilers, por serem, em geral, ferramentas mais completas para esse propósito, tendo suporte para várias métricas de desempenho, incluindo monitoramento de memória. Foram feitas buscas na Internet e em trabalhos científicos com o objetivo de identificar as ferramentas de análise estática e dinâmica com suporte para a linguagem Kotlin. Em seguida, foi feita uma análise da documentação dessas ferramentas com o objetivo de entender o que a ferramenta é capaz de analisar e identificar.

Dessa forma, foram selecionadas as ferramentas ESBMC-Jimple e Intellij Idea Inspector, que são ferramentas que possuem ênfase em detecção de problemas de segurança de memória, além das ferramentas JProfiler e Android Profiler, que possuem ênfase em análise de desempenho.

4.2.1 Descrição das Ferramentas Selecionadas

4.2.1.1 ESBMC-Jimple

O ESBMC-Jimple (MENEZES et al., 2022) é um verificador de modelos para verificar programas Kotlin por meio da representação intermediária Jimple. Ele é construído sobre o Efficient SMT-Based Context-Bounded Model Checker (ESBMC) e verifica as propriedades de segurança, como estouro aritmético, limites de matriz, divisão por zero e propriedades especificadas pelo usuário.

A Figura 2 mostra uma visão geral da arquitetura do ESBMC-Jimple, que pode ser descrita da seguinte forma:

Começando pelo lado esquerdo, temos como entra um arquivo .kt, que em seguida é traduzido para a representação intermediária Jimple. Na sequência, o código Jimple é transformado para o formato de uma AST .JSON, que então entra no Verificador BMC, que faz a verificação da satisfatibilidade. No caso de violação de propriedade, o programa gera um contraexemplo com detalhes da violação.

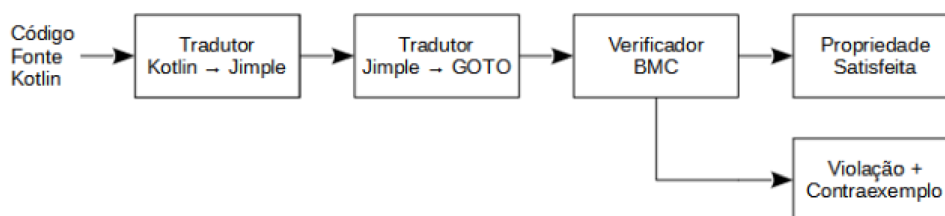

Figura 2 – Diagrama de blocos mostrando a visão geral da arquitetura do ESBMC-Jimple.

Fonte: Própria.

4.2.1.2 Intellij Idea Inspector

É uma ferramenta de análise estática de código disponível no ambiente de desenvolvimento IntelliJ IDEA (JetBrains, 2023). Ele é projetado para ajudar os desenvolvedores a encontrar e corrigir problemas de código em tempo real, melhorando a qualidade do código e a produtividade do desenvolvedor.

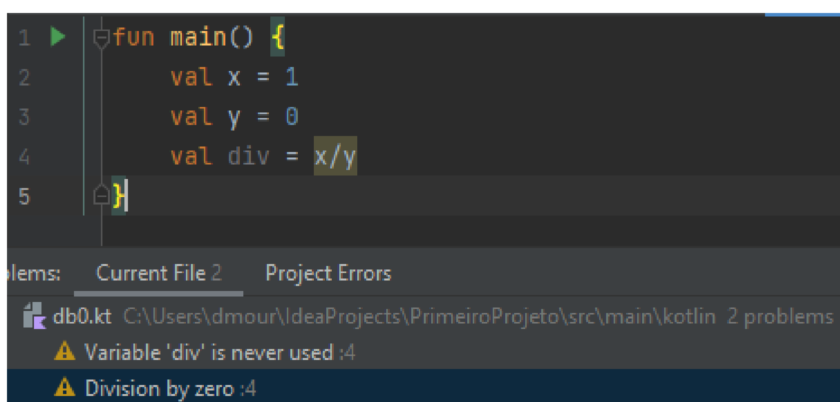

Figura 3 – Captura de tela do Intellij Idea com aba do Inspector reportando problemas no código.

Fonte: Própria.

A Figura 3 mostra o Inspector em ação, destacando o trecho do código contendo uma vulnerabilidade e reportando na aba selecionada na parte de baixo.

O Inspector tem suporte para várias linguagens que rodam na JVM.

4.2.1.3 JProfiler

JProfiler (ej-technologies GmbH, 2023) é uma ferramenta profissional para analisar programas executados na Java Virtual Machine (JVM). é amplamente usado pela

comunidade de desenvolvedores Java e possui muitos recursos. Agora também possui recursos específicos para a linguagem Kotlin, como corrotinas. Fornece informações de consumo de memória, atividade do Garbage Collector (GC), uso da CPU, entre outras métricas de desempenho.

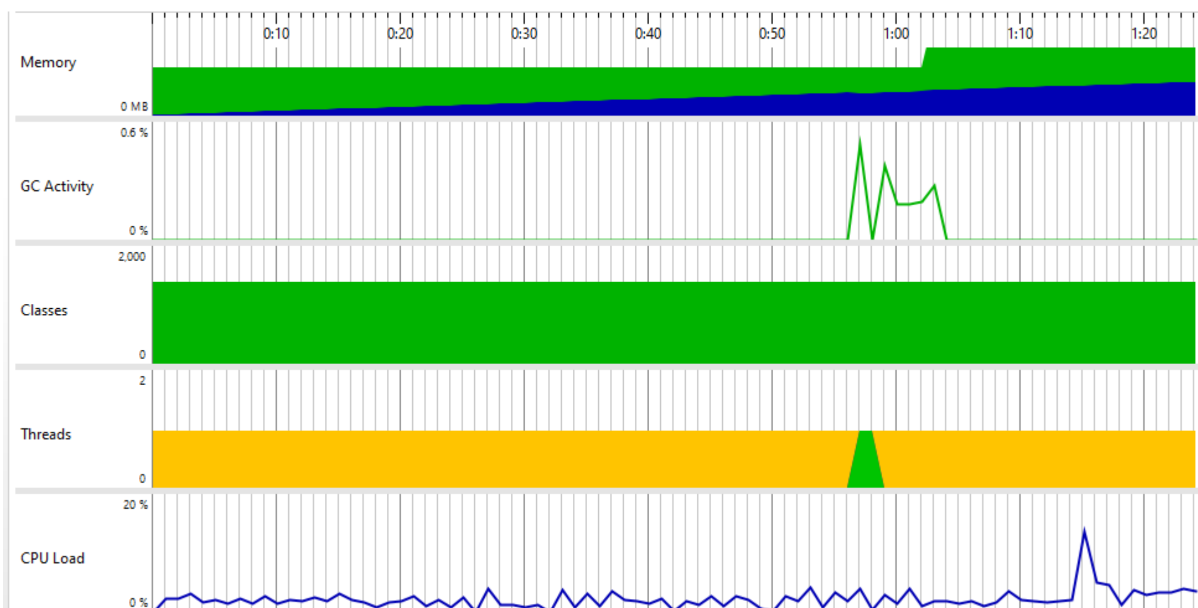

Figura 4 – Captura de tela mostrando o JProfiler em atividade.

Fonte: Própria.

Com o JProfiler também é possível visualizar uma tabela com os número de objetos criados durante a execução do programa e a memória consumida por cada um, o que pode ajudar a detectar vulnerabilidades, como vazamentos de memória. Pode ser integrado com IDEs Intellij Idea, Eclipse e Netbeans.

4.2.1.4 Android Profiler

O Android Profiler (Google, 2023) é uma ferramenta de criação de perfil que usa análise dinâmica para monitorar memória, CPU, rede e energia em aplicativos Android. Ele vem integrado ao IDE Android Studio.

A Figura 5 mostra o Android Profiler em atividade, mostrando dados referentes à diversas métricas de desempenho através de gráficos.

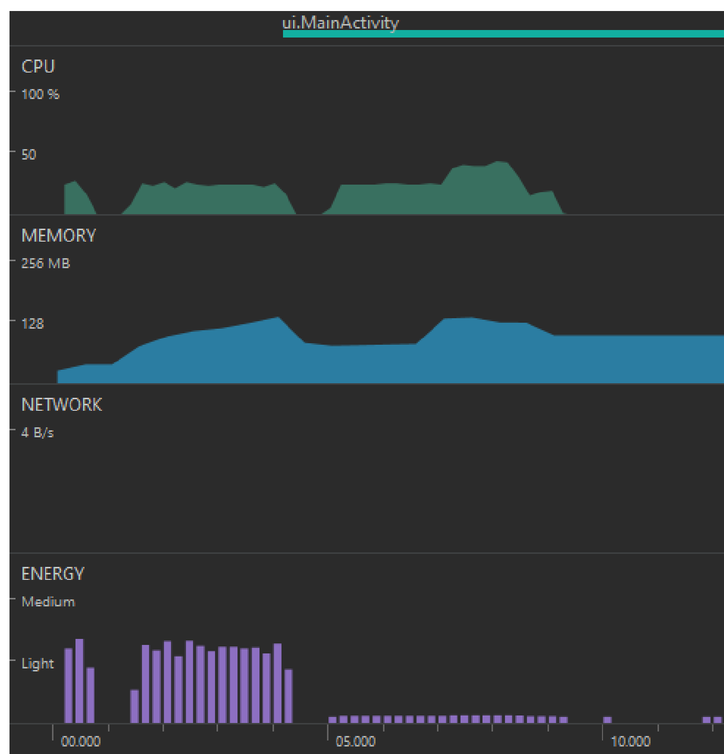

Figura 5 – Captura de tela mostrando o Android Profiler em atividade.

Fonte: Própria.

4.3 Avaliação Experimental

Para o estudo, foram realizados dois experimentos principais: um dos experimentos contém uma análise comparativa entre ferramentas de análise estática e dinâmica com suporte para a linguagem Kotlin e o outro consiste em um estudo empírico utilizando essas ferramentas com o objetivo de observar possíveis implicações diretas de problemas de segurança de memória no desempenho de programas Kotlin. Na presente seção, descrevemos os objetivos e configurações gerais dos experimentos, as *benchmarks* utilizadas e como se deu o desenvolvimento de cada experimento.

4.3.1 Configurações Gerais dos Experimentos

Os experimentos são feitos em um laptop com Sistema Operacional Windows 10, com 16 GB de memória RAM e um processador Intel Core I7 da 11ª geração. Configurações específicas de cada experimento estão descritas nas suas relativas subseções.

4.3.2 Descrição das *benchmarks*

Para a avaliação experimental, foram utilizadas as seguintes *benchmarks*:

- *StackOFlow.kt*: possui uma chamada recursiva sem condição de parada que ocasiona um estouro de pilha;
- *fatorial_oflow.kt*: programa para calcular o fatorial de um número, porém com um problema de implementação que leva a uma chamada recursiva profunda que leva a um estouro de pilha;
- *db0.kt*: possui uma divisão por zero;
- *int_oflow.kt*: possui um estouro aritmético de inteiro;
- *array_oob.kt*: possui uma violação nos limites de um array.
- *ListaDeObjetos.kt*: simples programa que contém uma infinita adição de objetos em uma lista.
- *ListaDeObjetosLimitada.kt*: programa com adição de objetos em uma lista, porém com limite.
- Aplicativo *Simple Notes*: Simples aplicativo de notas para Android escrito com linguagem Kotlin. Disponível em (DJSMK123, 2023).

O código das *benchmarks* utilizadas estão disponíveis no Apêndice A. Algumas das benchmarks foram retiradas do conjunto de *benchmarks* do artigo *ESBMC-Jimple: Verifying Kotlin Programs Using Jimple Intermediate Representation* (MENEZES et al., 2022).

4.3.3 Análise das Ferramentas de Detecção de Problemas de Segurança de Memória

As ferramentas ESBMC-Jimple e Intellij Idea Inspector foram testadas contra benchmarks contendo problemas de estouro de pilha (em inglês, *Stack Overflow*) no CT3 e CT4, Divisão por Zero no CT1, estouro de inteiros (em inglês, *Integer Overflow*)

no CT0 e violação de limites de um array (em inglês, *Array Bounds Violation*) no CT2 e seus desempenhos foram avaliados de acordo com as seguintes questões experimentais (QE):

(QE1)- A ferramenta pode identificar corretamente um determinado problema de segurança de memória?

(QE2)- A ferramenta fornece um contraexemplo contendo detalhes do problema encontrado?

Para os testes com a ferramenta ESBMC-Jimple, primeiramente utilizamos o framework Soot 4.3.0 (TEAM, 2023) para transformar o arquivo .kt para a representação intermediária Jimple utilizando os três comandos abaixo:

kotlinc -include-runtime -d output.jar <file>.kt

jar xf output.jar

java -cp .-4.3.0-jar-with-dependencies.jar soot.Main -cp . -pp -f jimple OriginalKt -write-local-annotations -p jb use-original-names:true -keep-line-number -print-tags-in-output

Após a obtenção do arquivo .jimple, o mesmo deve passar por um outro módulo (SAMENEZES, 2023) que faz a conversão do arquivo para o formato .json.

No ESBMC-Jimple, para fazer a checagem de Divisão por Zero e Violação de Limites de *Array*, não é necessária nenhuma flag adicional.

Para checar um estouro de inteiros, é necessário adicionar a flag *–overflow-check*.

Para checar um estouro de pilha, é necessário acrescentar a flag *–stack-limit <valor>*. Como a benchmark relativa à esse experimento possui uma recursão sem condição de parada, foi necessário também acrescentar a flag *–incremental-bmc*, que aciona a estratégia de verificação *incremental bmc*.

O Intellij Idea Inspector faz a análise do código-fonte no próprio editor, durante a escrita do código, e reporta os problemas encontrados na aba *Problems*.

Resultados da análise comparativa das ferramentas:

A Tabela 2 mostra os resultados da análise comparativa das ferramentas de detecção de problemas de segurança de memória.

A coluna "Detectou" indica o encontro de um bug, seguido por uma coluna "CE" mostrando se um contraexemplo foi gerado. As duas últimas linhas mostram a

Tabela 2 – Resultados da análise comparativa entre as ferramentas de detecção de problemas de segurança de memória .

Benchmark		Intellij Idea Inspector		ESBMC-Jimple	
IDs	**Vulnerabilidade**	**Detectou**	**CE**	**Detectou**	**CE**
CT0	Int-Overflow	Não	Não	Sim	Sim
CT1	Div-por-zero	Sim	Sim	Sim	Sim
CT2	Out-of-bounds	Não	Não	Sim	Sim
CT3	Stack-Overflow	Não	Não	Sim	Sim
CT4	Stack-Overflow	Não	Não	Sim	Sim
Resultados Corretos		20%		100%	
Resultados Confirmados		20%		100%	

Fonte: Própria.

porcentagem dos resultados corretos e os confirmados, sendo esses resultados usados como base para responder as questões experimentais **QE1** e **QE2**. Os resultados mostram que a ferramenta ESBMC-Jimple teve o melhor desempenho no que diz respeito a detecção de problemas de segurança de memória em programas Kotlin, sendo capaz de detectar erros de memória em todas as *benchmarks*. O Intellij Idea Inspector faz uma análise mais superficial, a nível de código-fonte, e conseguiu identificar apenas um erro de Divisão por Zero, relativo ao CT1.

4.3.4 Análise das Ferramentas de Monitoramento de Desempenho

Para a análise das ferramentas com ênfase em monitoramento de desempenho, foi feita uma avaliação e comparação das principais funcionalidades das ferramentas JProfiler e Android Profiler. A análise é qualitativa e busca explorar as funcionalidades de cada ferramenta.

Devido à dificuldade de encontrar programas Kotlin compatíveis com ambos os profilers pela diferença entre as máquinas virtuais, utilizaram-se *benchmarks* diferentes, por isso não foram explorados aspectos como precisão das medidas.

A Tabela 3 mostra algumas características gerais das ferramentas JProfiler e Android Profiler.

Tabela 3 – Características gerais do JProfiler e do Android Profiler.

Característica	**JProfiler**	**Android Profiler**
Plataforma de suporte	JVM	Android
Tipos de Aplicativos Suportados	Java e Kotlin	Android
Técnicas de coleta de dados	Instrumentação e amostragem	Instrumentação e amostragem

Fonte: Própria.

Para a análise do JProfiler, foi usada a benchmark *ListaDeObjetos.kt*, já descrita na seção de descrição das benchmarks e disponível no Apêndice A. Para a análise do Android Profiler, foi usado o aplicativo *Simple Notes*. Foram usadas as versões JProfiler v12.0.4 e Android Studio Arctic Fox 2020.3.1.

Foram exploradas as capacidades de ambas as ferramentas de coletar dados relativos às métricas de consumo de memória, uso de CPU e atividade do coletor de lixo, além de recursos adicionais que cada uma possui.

Resultados da análise comparativa das ferramentas:

Como visto na Figura 6, o JProfiler fornece um gráfico com informações da memória Heap, podendo mostrar o consumo de memória dos objetos de cada geração (Eden, Old e Survivor). Mostra uma legenda identificando cada aspecto do gráfico, como espaço livre, espaço usado e espaço total. Além disso, mostra uma lista contendo contagem das instâncias das classes e a memória consumida por cada uma.

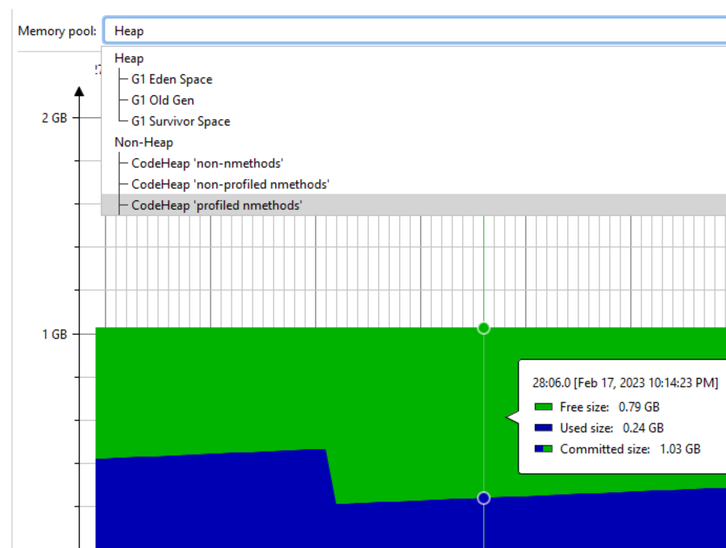

Figura 6 – Captura de tela do monitoramento de memória do JProfiler.
Fonte: Própria.

O Android Profiler apresenta um gráfico de monitoramento de memória com várias camadas, que representam o espaço de memória ocupado por diferentes classes de objetos, como objetos de linguagem nativa, objetos Java/Kotlin, entre outras classes comumente presentes em aplicativos Android, como visto na Figura 7.

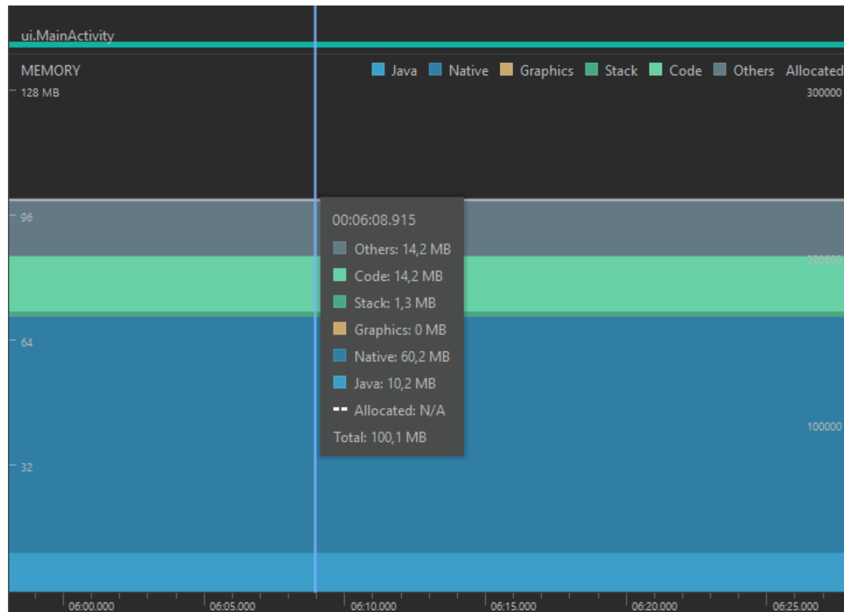

Figura 7 – Captura de tela do monitoramento de memória no Android Profiler.
Fonte: Própria.

O JProfiler apresenta um gráfico mostrando informações de uso da CPU total do sistema e também do processo específico em porcentagem, como visto na Figura 8.

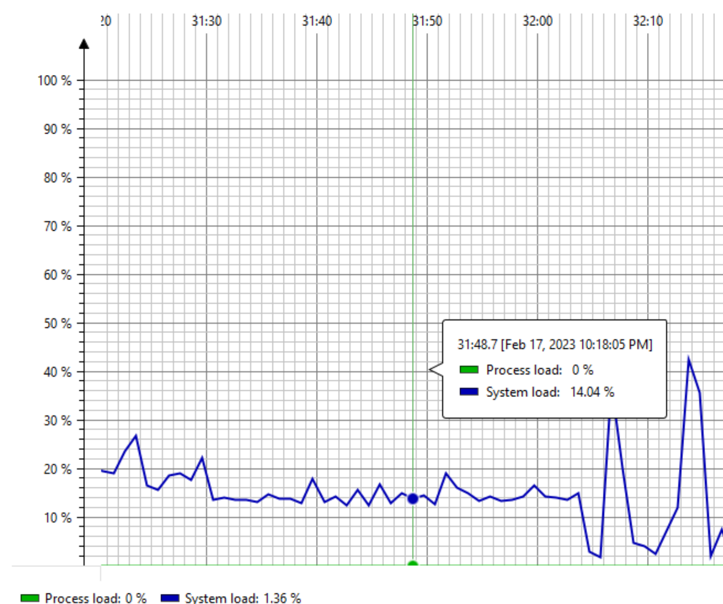

Figura 8 – Captura de tela do monitoramento de CPU do JProfiler.
Fonte: Própria.

O Android Profiler também possui suporte para coleta de dados referente ao uso da CPU, como mostra a Figura 9.

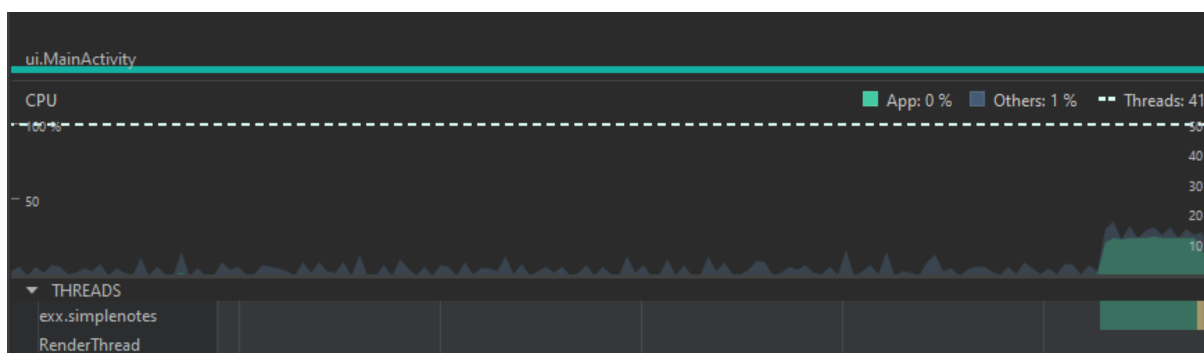

Figura 9 – Captura de tela do monitoramento de uso de CPU no Android Profiler. Fonte: Própria.

O JProfiler apresenta um gráfico com a atividade do coletor de lixo ao longo do tempo, como mostra a Figura 10 .

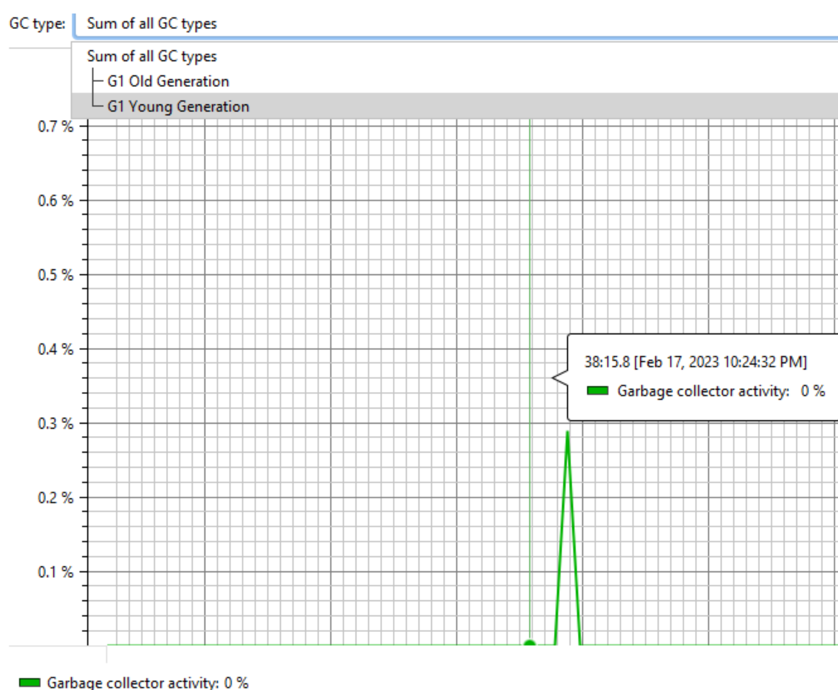

Figura 10 – Captura de tela mostrando a atividade do coletor de lixo no JProfiler. Fonte: Própria.

O Android Profiler também pode acompanhar a atividade do coletor de lixo, como visto na Figura 11.

O JProfiler possui informações detalhadas sobre alocações de objetos, contendo informações sobre o número de alocações e o espaço ocupado pelas instâncias, além de ordenar e apontar as classes com mais instâncias criadas e as que estão ocupando mais

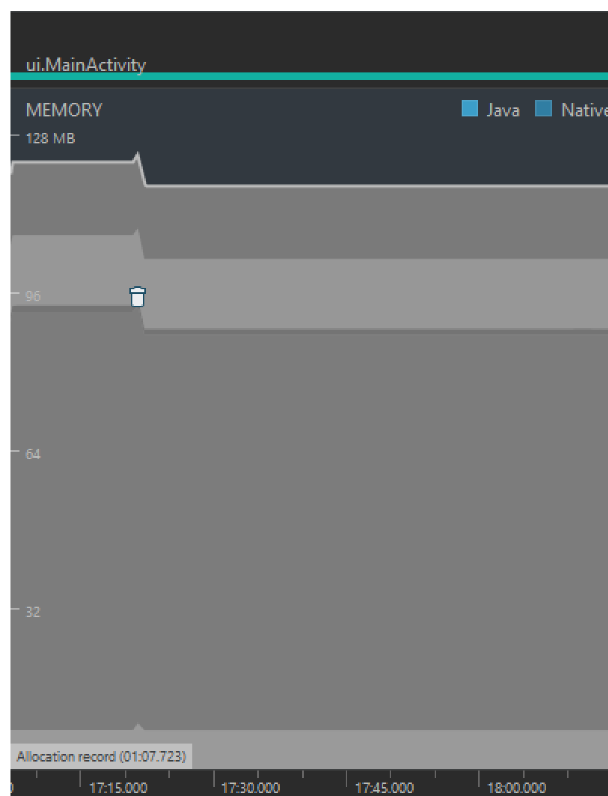

Figura 11 – Captura de tela mostrando a atividade do coletor de lixo no Android Profiler.

Fonte: Própria.

espaço na memória. O Android Profiler mostra informações de alocações de objetos no mesmo gráfico onde mostra o perfil de memória.

O JProfiler tem capacidade de mostrar pilhas de chamadas, indicando também o tempo de cada chamada. Pode identificar, por exemplo, recursões muito profundas. O Android Profiler não possui esse recurso.

O JProfiler analisa o fluxo de execução de um programa através da função *call tree*, podendo representar graficamente hierarquias de chamadas de métodos que foram executadas pelo programa. O Android Profiler não possui esse recurso.

Ambos podem capturar informações para analisar o consumo de recursos em tempos específicos da execução, recurso conhecido em inglês como snapshot.

É difícil fazer uma comparação mais clara e justa entre os profilers pois, apesar de ambos suportarem a linguagem Kotlin, o Android Profiler acaba sendo de uso mais específico para o Android, dessa forma suas funcionalidades foram projetadas para atender a esse propósito. O JProfiler possui mais funcionalidades por analisar programas Kotlin de propósito mais geral.

4.3.5 Estudo Empírico Investigando Relações Entre Problemas de Segurança de Memória e Desempenho em Programas Kotlin

O estudo empírico a seguir é composto de dois experimentos utilizando as ferramentas analisadas para identificar problemas de segurança de memória em programas Kotlin e verificar a possibilidade de identificar gargalos de desempenho nesses programas. O Experimento 1 analisa a benchmark *StackOFlow.kt*, que contém uma função com chamadas recursivas sem uma condição de parada, que em todos os casos deve resultar em um erro de estouro de pilha. O Experimento 2 analisa a *benchmark ListaDeObjetos.kt*, que possui uma lista com adição de objetos de forma ininterrupta, o que leva a um padrão de consumo de memória excessivo, além da *benchmark ListaDeObjetosLimitada.kt*, que limita a lista. Para o estudo, foram selecionadas as ferramentas ESBMC-Jimple, capaz de identificar e reportar estouros de pilha em programas Kotlin, e JProfiler, capaz de observar a sequência de chamadas recursivas que leva ao estouro da pilha, além de gerar gráficos com informações de consumo de memória e coleta de lixo.

4.3.5.1 Experimento 1: Analisando um programa com estouro de pilha

O Experimento 1 utiliza as ferramentas de análise estática e dinâmica com suporte para a linguagem Kotlin para analisar com mais profundidade um problema de segurança de memória e verificar se é possível observar possíveis implicações diretas no desempenho do programa.

Para o experimento foi utilizada a ferramenta ESBMC-Jimple para identificar o estouro da pilha da benchmark StackOFLow.kt através de análise estática. O tamanho da pilha foi aumentado e adicionado um package, para que o mesmo pudesse ser monitorado pelo JProfiler.

Primeiramente executamos o experimento com o tamanho de memória padrão da pilha da JVM, e posteriormente aumentamos o tamanho para 10Mb. Na função Call Tracer do JProfiler, as chamadas de função foram gravadas por tempos diferentes, para que possamos observar o crescente número de chamadas recursivas que tendem a esgotar a memória da pilha. Por último, verificamos se é possível observar as consequências

do estouro da pilha e se há implicações no desempenho do programa.

4.3.5.2 Resultados e Discussão do Experimento 1

Os resultados mostram que, para que as chamadas recursivas pudessem ser observadas com o JProfiler, foi necessário aumentar o tamanho da pilha, visto que o limite padrão da pilha é atingido muito rápido e dessa forma o tempo de execução da benchmark é muito curto, não sendo o suficiente para que o JProfiler pudesse fazer a leitura.

Aumentando o limite da pilha através da flag -Xss10m, aumentando o tamanho da pilha para 10Mb, o JProfiler foi capaz de fazer a leitura, mostrando as múltiplas chamadas de função e fornecendo o tempo de cada chamada, como visto na Figura 12

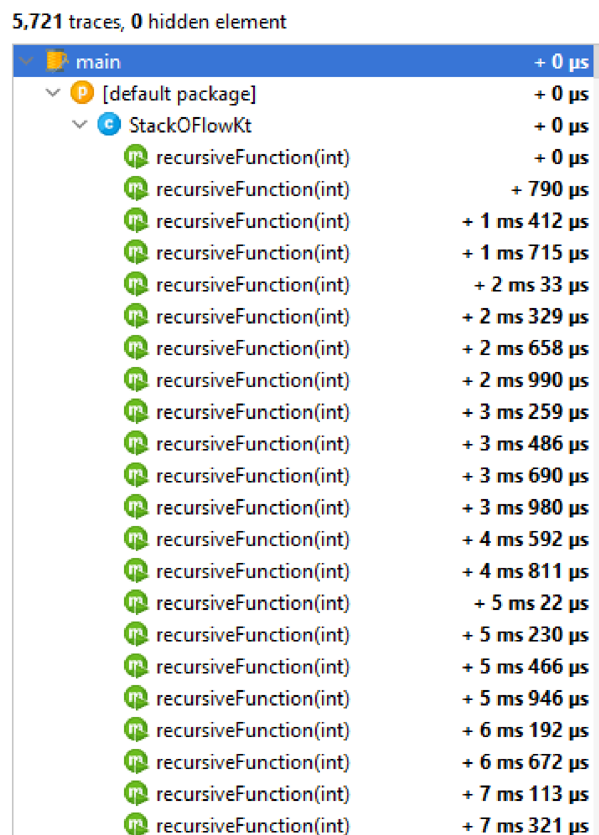

Figura 12 – Call tracer do JProfiler mostrando as múltiplas chamadas da função recursiva.

Fonte: Própria.

O call tracer apresenta o número de chamadas realizados pela função e o tempo de cada chamada.

Tabela 4 – Tabela mostrando o número de chamadas recursivas da função *recursiveFunction()* em cada espaço de tempo de captura.

Tempo de captura em segundos	**Número de chamadas**
≈ 2	5721
≈ 4	9807
≈ 6	16662
≈ 8	21339
≈ 10	28227

Fonte: Própria.

A Tabela 4 mostra o tempo de cada captura e o número de chamadas recursivas feitas no relativo espaço de tempo, mostrando o crescimento no número de chamadas recursivas que tendem a levar ao esgotamento da memória da pilha

O JProfiler foi capaz de mostrar a imensa pilha de chamadas geradas em espaços de tempo diferentes, mas tem dificuldades em detectar o exato momento do estouro da pilha devido ao término do programa, quando foi lançada a exceção *java.lang.StackOverflowError*. Além disso, devido ao término da execução do programa pela JVM ao detectar o estouro da pilha, não foi possível observar o que acontece após o estouro da pilha, não sendo possível observar possíveis implicações no desempenho do programa. No caso, a ferramenta ESBMC-Jimple pode ser útil ao identificar o estouro da pilha antes da execução do programa, visto que, caso exista a possibilidade da JVM não identificar o erro através do sistema de exceções, a continuidade do programa após o estouro pode levar a graves problemas.

4.3.5.3 Experimento 2: programa com consumo crescente e ininterrupto de memória

Para o segundo experimento, foi utilizada a ferramenta JProfiler, que analisou a benchmark *ListaDeObjetos.kt*, que contém uma adição ininterrupta de elementos em uma lista, sem uma instrução para retirar os objetos da lista. Ao iniciar o teste, foram analisados os gráficos de consumo de memória e de atividade do coletor de lixo, com o objetivo de observar a curva de crescimento dos objetos alocados e a frequência da atividade do GC. Além disso, foi utilizada a função Heap Walker, que auxilia na

localização das classes que possuem mais objetos alocados do programa em execução e quanto espaço de memória estão ocupando. A memória heap máxima reservada para o programa é de 4225.8MB.

Em seguida, foi analisada a benchmark *ListaDeObjetosLimitada.kt* (ver Apêndice A), que se diferencia da primeira apenas pela definição de um limite de objetos para a lista. No caso, o limite máximo de objetos para a lista é de 100 objetos. Após isso, o objeto mais antigo é removido antes de adicionar o novo objeto.

4.3.5.4 Resultados e Discussão do Experimento 2

Os resultados mostram o crescente número de objetos alocados na memória heap ao longo do tempo na benchmark *ListaDeObjetos.kt*, ocupando 102.2MB de memória em aproximadamente 46 segundos, como visto na Figura 13 .

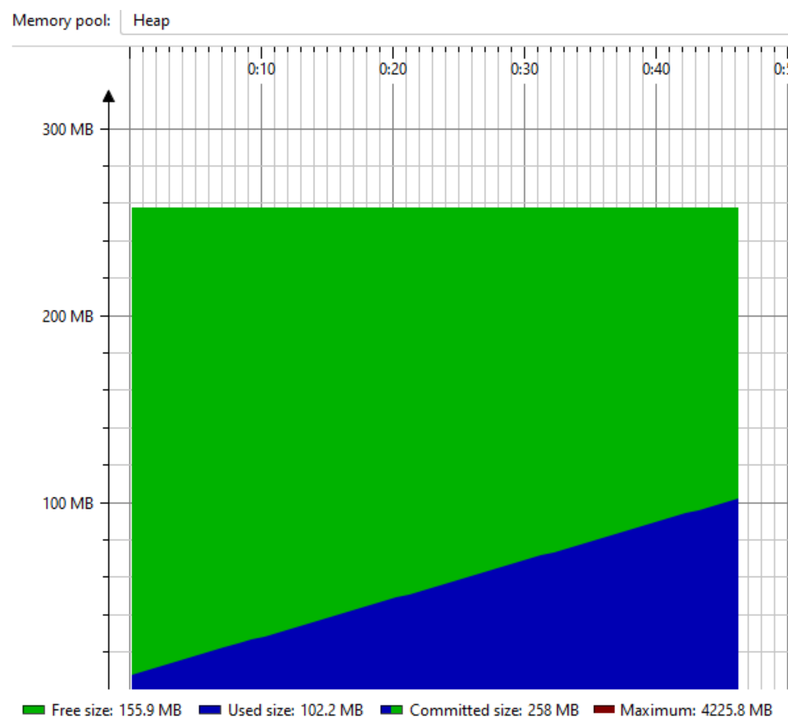

Figura 13 – Trace de memória mostrando o consumo crescente de memória heap.
Fonte: Própria.

No início da execução, o espaço de memória total reservada para o programa é de 258MB. Com o passar do tempo, o coletor de lixo começa as suas primeiras atividades, como pode ser visto na Figura 14.

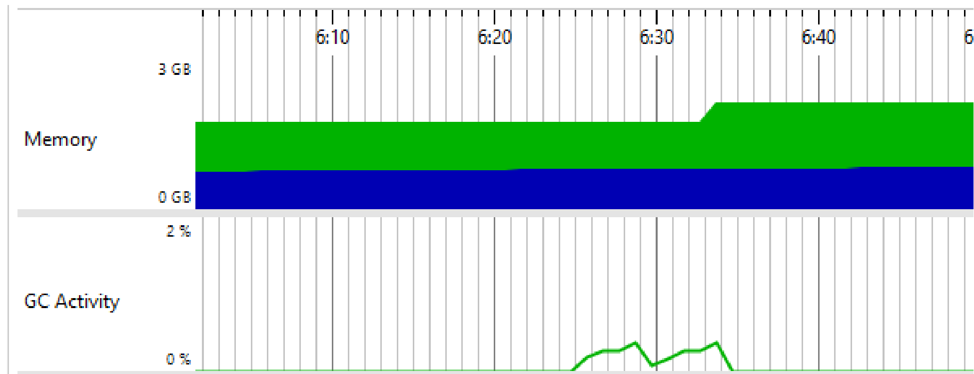

Figura 14 – Tela mostrando o trace de memória e a atuação do coletor de lixo.
Fonte: Própria.

Pode-se notar também que, junto com a atuação do coletor de lixo, ocorre um aumento da memória heap livre, representada pela área verde.

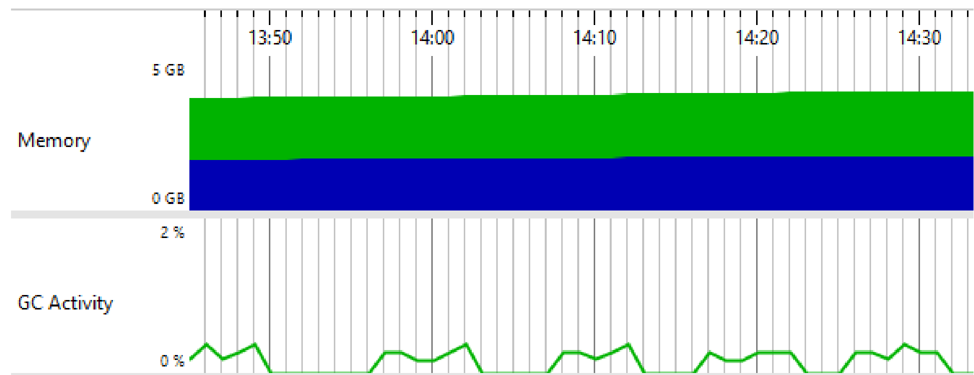

Figura 15 – Tela mostrando o coletor de lixo em frequente atividade.
Fonte: Própria.

Na Figura 15, podemos ver um momento em que o coletor de lixo começa a ter frequentes atuações, com aproximadamente 10 segundos de intervalo entre cada atuação. Além disso, o limite máximo de memória heap que o programa disponibiliza já está próximo de ser atingido (4225.8MB).

Current object set: 34,212 objects in 644 classes.

1 selection step, 705 MB shallow size

Classes | Use ... | Group By Class Loaders

Name	Instance Count	Size	Retained Size
byte[ ]	8,176	704 MB	704 MB
java.lang.String	7,316	175 kB	≥ 492 kB
java.util.concurrent.ConcurrentHashMap$Node	2,238	71,616 bytes	≥ 133 kB
java.lang.Object[ ]	1,967	128 kB	≥ 703 MB
java.lang.Class	1,592	509 kB	≥ 844 kB
java.util.HashMap$Node	1,264	40,448 bytes	≥ 68,784 bytes
Objeto	671	10,736 bytes	≥ 703 MB
java.util.HashMap	387	18,576 bytes	≥ 102 kB
java.lang.String[ ]	377	7,616 bytes	≥ 9,440 bytes
java.lang.module.ModuleDescriptor$Exports	362	8,688 bytes	≥ 11,248 bytes
Total:	34,212	705 MB	

Figura 16 – Número de instâncias e tamanho ocupado pela Classe Objeto (em destaque).
Fonte: Própria.

Na Figura 16, podemos ver a função Heap Walker mostrando a Classe Objeto ocupando grande espaço na memória do programa, com 671 instâncias criadas e o tamanho total ocupado maior ou igual a 703MB.

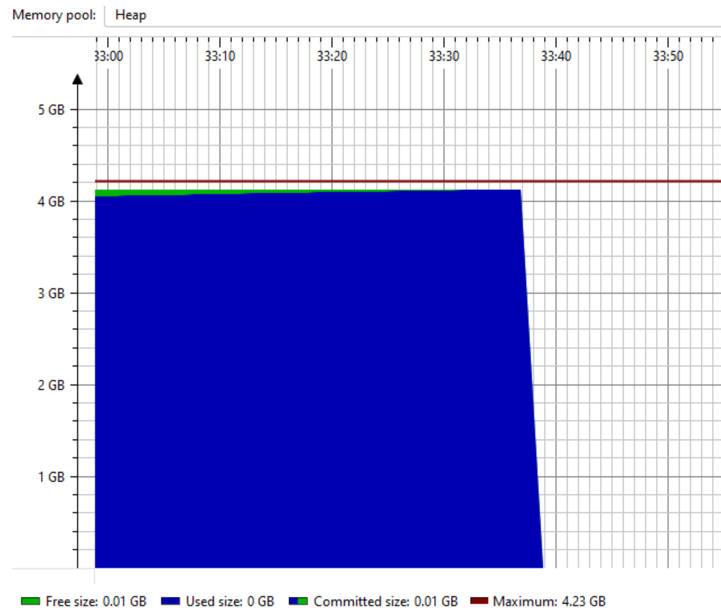

Figura 17 – Momento em que o limite de memória é atingido.
Fonte: Própria.

Na Figura 17, podemos ver o momento em que o programa atinge o limite de memória reservado e o programa é interrompido após um pouco mais de 33 minutos de execução após a exceção java.lang.OutOfMemoryError ser lançada.

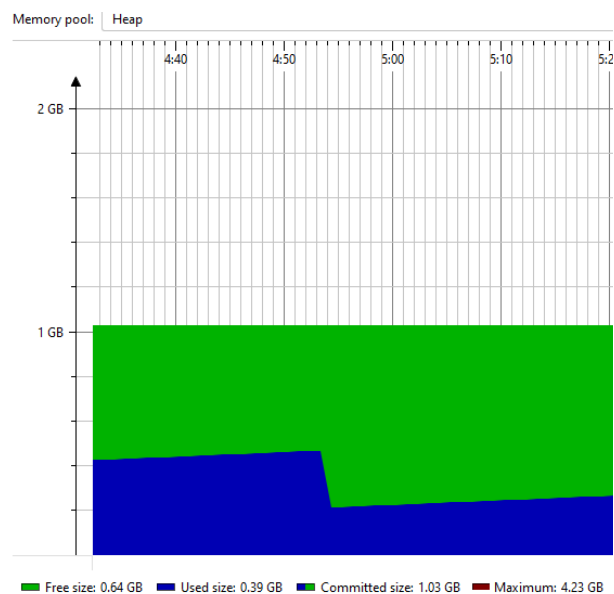

Figura 18 – Consumo de memória no programa *ListaDeObjetosLimitada*.
Fonte: Própria.

Na Figura 18, podemos ver o consumo de memória heap no caso em que a lista de objetos é limitada e podemos ver uma melhora no gerenciamento dos recursos de memória.

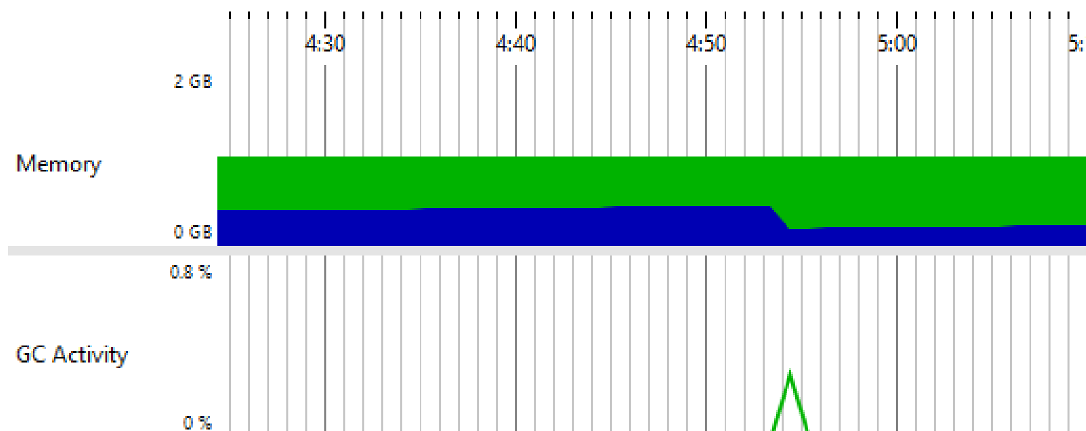

Figura 19 – Momento em que o coletor de lixo atua no programa *ListaDeObjetosLimitada*.
Fonte: Própria.

A Figura 19 mostra o momento em que o coletor de lixo atua, conseguindo liberar a memória dos objetos que já não estão mais na lista.

A partir dos resultados, é possível notar como o JProfiler pode analisar possíveis gargalos de desempenho relativos a um problema de segurança de memória. O monitoramento da memória heap juntamente com o monitoramento da atividade do coletor de lixo já podem nos dar grandes indícios de problemas de desempenho.

A atividade frequente do coletor de lixo também traz sobrecarga ao sistema e é um outro elemento que pode afetar o desempenho de um programa.

No caso com a lista de objetos limitada, pudemos ver que o consumo de memória heap pôde ser controlado com a ajuda do coletor de lixo, que não teve sobrecarga.

Apesar de ter sido possível observar o problema de desempenho, a detecção do problema de segurança de memória foi feito de forma manual. Dessa forma, não foi possível combinar as ferramentas nesse caso também.

4.4 Ameaças à validade

- Algumas ferramentas que possuem suporte para análise de *bytecode* não foram consideradas ou selecionadas, pois suas documentações não mencionavam suporte

para Kotlin. No entanto, existe a possibilidade de algumas dessas ferramentas suportarem Kotlin, apesar de a linguagem ter algumas estruturas específicas.

- Houve dificuldade em encontrar sets de *benchmarks* específicas para avaliar ferramentas de detecção de problemas de segurança de memória e problemas de desempenho em programas Kotlin, por isso foi utilizado um set de *benchmarks* pequeno para os experimentos, que também foi baseado no que a documentação das ferramentas afirma suportar relacionado a problemas de segurança de memória e desempenho.

5

Conclusões

O estudo pretendeu entender desempenho em programas Kotlin sob uma perspectiva de segurança de memória através de um estudo empírico utilizando ferramentas de análise estática e dinâmica. Para isso, identificou e descreveu alguns dos problemas mais graves e comuns relacionados à segurança de memória, selecionou e analisou as ferramentas e, ao final, buscou formas de utilizá-las para observar possíveis implicações no desempenho relacionados a esses problemas. O ESBMC-Jimple apresentou os melhores resultados na detecção de problemas de segurança de memória, e o JProfiler foi a ferramenta que apresentou mais recursos para análise de desempenho de programas Kotlin em geral. Tais ferramentas foram selecionadas para o estudo empírico que buscou observar implicações de problemas de segurança de memória no desempenho dos programas. Em um dos experimentos, o ESBMC-Jimple identificou um estouro de pilha, mas não foi possível observar suas consequências devido à interrupção do programa pela JVM. No outro, foi possível observar com o JProfiler um consumo excessivo de memória em um programa que levou a um esgotamento desse recurso, porém só foi possível identificar o problema de segurança de memória através de revisão manual do código. Dessa forma, não foi possível observar implicações de problemas de segurança de memória no desempenho de programas Kotlin combinando as ferramentas selecionadas para o estudo. Notou-se a necessidade de identificação dos problemas de segurança de memória mais recorrentes em programas Kotlin e uma pesquisa para identificação de ferramentas com suporte para detecção desses tipos de problemas. O estudo utilizou ferramentas de análise estática e dinâmica para explorar as fronteiras entre os problemas

de segurança de memória e desempenho e acabou abrindo a possibilidade de estudos a respeito de variadas áreas da qualidade de software.

5.1 Trabalhos Futuros

Uma possibilidade de estudo a ser feito a seguir seria um levantamento e descrição das vulnerabilidades relacionadas a problemas de segurança mais frequentemente encontrados em programas Kotlin.

Referências

ALBREIKI, H. H.; MAHMOUD, Q. H. Evaluation of static analysis tools for software security. In: *2014 10th International Conference on Innovations in Information Technology (IIT)*. [S.l.: s.n.], 2014. p. 93–98. 27

ARTIFACTS. *Detekt Kotlin Static Code Analysis*. 2021. Acesso em: 10 de Fevereiro de 2023. Disponível em: <https://detekt.github.io/detekt/>. 26

BAIER, C.; KATOEN, J.-P. Principles of model checking. In: . [S.l.: s.n.], 2008. 26

BODDEN, E. Inter-procedural data-flow analysis with ifds/ide and soot. In: . [S.l.: s.n.], 2012. p. 3–8. 26

COFFIN, D. Expert oracle and java security. 01 2011. 24

CORDEIRO, L. et al. Jbmc: A bounded model checking tool for verifying java bytecode: 30th international conference, cav 2018, held as part of the federated logic conference, floc 2018, oxford, uk, july 14-17, 2018, proceedings, part i. In: ____. [S.l.: s.n.], 2018. p. 183–190. ISBN 978-3-319-96144-6. 26

CORDEIRO, L. et al. Context-bounded model checking with esbmc 1.17. In: . [S.l.: s.n.], 2012. p. 534–537. ISBN 978-3-642-28755-8. 26

CORDEIRO, L. C. Exploiting the SAT revolution for automated software verification: Report from an industrial case study. In: *10th Latin-American Symposium on Dependable Computing, LADC 2021, Florianópolis, Brazil, November 22-26, 2021 - Companion Volume*. Brazilian Computing Society, 2021. p. 8–9. Disponível em: <https://doi.org/10.5753/ladc.2021.18531>. 15

CORDEIRO, L. C.; FILHO, E. B. de L.; BESSA, I. V. de. Survey on automated symbolic verification and its application for synthesising cyber-physical systems. *IET Cyper-Phys. Syst.: Theory & Appl.*, v. 5, n. 1, p. 1–24, 2020. Disponível em: <https://doi.org/10.1049/iet-cps.2018.5006>. 16

CORDEIRO, L. C.; KROENING, D.; SCHRAMMEL, P. JBMC: bounded model checking for java bytecode - (competition contribution). In: BEYER, D. et al. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*. Springer, 2019. (Lecture Notes in Computer Science, v. 11429), p. 219–223. Disponível em: <https://doi.org/10.1007/978-3-030-17502-3_17>. 26

DHURJATI, D. et al. Memory safety without runtime checks or garbage collection. *ACM SIGPLAN Notices*, v. 38, 11 2003. 23

DJSMK123. *Android-Study-Jams*. GitHub, 2023. Disponível em: <https://github.com/Djsmk123/Android-Study-Jams>. 37

ej-technologies GmbH. *JProfiler - Your Java profiler for all occasions*. 2023. Acesso em: 08 de Fevereiro de 2023. Disponível em: <https://www.ej-technologies.com/products/jprofiler>. 16, 26, 34

FLAIG, A.; HERTL, D.; KRÜGER, F. Fachstudie Softwaretechnik, *Evaluation von Java Profiler Werkzeugen*. 2013. 89 p. Fachstudie Softwaretechnik: Universität Stuttgart, Institut für Formale Methoden der Informatik, Sichere und Zuverlässige Softwaresysteme. Disponível em: <http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=FACH-0184&engl=0>. 29

FOUNDATION, K. *Kotlin Language*. 2022. Disponível em: <https://kotlinlang.org/>. 15, 21, 22, 23

GADELHA, M. R.; CORDEIRO, L. C.; NICOLE, D. A. An efficient floating-point bit-blasting API for verifying C programs. In: CHRISTAKIS, M. et al. (Ed.). *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers*. Springer, 2020. (Lecture Notes in Computer Science, v. 12549), p. 178–195. Disponível em: <https://doi.org/10.1007/978-3-030-63618-0_11>. 25

GADELHA, M. R.; MENEZES, R. S.; CORDEIRO, L. C. ESBMC 6.1: automated test case generation using bounded model checking. *Int. J. Softw. Tools Technol. Transf.*, v. 23, n. 6, p. 857–861, 2021. Disponível em: <https://doi.org/10.1007/s10009-020-00571-2>. 16

GADELHA, M. Y. R. et al. ESBMC: scalable and precise test generation based on the floating-point theory - (competition contribution). In: WEHRHEIM, H.; CABOT, J. (Ed.). *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Springer, 2020. (Lecture Notes in Computer Science, v. 12076), p. 525–529. Disponível em: <https://doi.org/10.1007/978-3-030-45234-6_27>. 25

GANDHEWAR, N.; SHEIKH, R. Google android: An emerging software platform for mobile devices. *Journal of Computer Science and Engineering*, v. 1, p. 12–17, 01 2011. 15

GANGWAR, D. K.; KATAL, A. Memory leak detection tools: A comparative analysis. In: *2021 International Conference on Recent Trends on Electronics, Information, Communication Technology (RTEICT)*. [S.l.: s.n.], 2021. p. 315–320. 32

GCN. *Software Glitches Leave Navy Smart Ship Dead in the Water*. 1998. <https://gcn.com/1998/07/software-glitches-leave-navy-smart-ship-dead-in-the-water/290995/>. Acessado em: 18 de fevereiro de 2023. 32

Google. *Android Studio*. 2023. Acesso em: 08 de Fevereiro de 2023. Disponível em: <https://developer.android.com/studio>. 16, 26, 35

JetBrains. *JetBrains*. 2023. Acesso em: 08 de Fevereiro de 2023. Disponível em: <https://www.jetbrains.com/help/idea>. 16, 26, 34

JONES, J. R. Estimating software vulnerabilities. *IEEE Security Privacy*, v. 5, n. 4, p. 28–32, 2007. 23

KAHSAI, T. et al. Jayhorn: A framework for verifying java programs. In: . [S.l.: s.n.], 2016. v. 9779, p. 352–358. ISBN 978-3-319-41527-7. 26

KHAN, A.; KUCHERENKO, I. *Hands-On Object-Oriented Programming with Kotlin: Build robust software with reusable code using OOP principles and design patterns in Kotlin*. Packt Publishing, 2018. ISBN 9781789619645. Disponível em: <https://books.google.com.br/books?id=buh1DwAAQBAJ>. 15, 21, 22

LAM, L.; CHIUEH, t.-c. Checking array bound violation using segmentation hardware. In: . [S.l.: s.n.], 2005. p. 388–397. 32

LEAL, B. G. *Análise e Projeto de Sistemas*. Petrolina, Brazil: Editora da Universidade Federal do Vale do São Francisco, 2015. Disponível em: <http://www.univasf.edu.br/~brauliro.leal/livro/ADS.pdf>. 25

MAHMOOD, R.; MAHMOUD, Q. *Evaluation of Static Analysis Tools for Finding Vulunerbailities in Java and C/C++ Source Code*. 2018. 28, 31

MARTINS, H. G. *Estudo Sobre a Exploração de Vulnerabilidades Via Estouros de Buffer, Sobre Mecanismos de Proteção e Suas Fraquezas*. 70 p. Monografia (Graduação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Rio Grande do Sul, 2009. 16, 32

MENEZES, R. et al. Esbmc-jimple: verifying kotlin programs via jimple intermediate representation. In: . [S.l.: s.n.], 2022. p. 777–780. 16, 31, 33, 37

MITRE Corporation. *Common Weakness Enumeration (CWE)*. 2023. Acesso em: 14 de Fevereiro de 2023. Disponível em: <https://cwe.mitre.org/>. 31

MONTEIRO, F. R.; GADELHA, M. R.; CORDEIRO, L. C. Summary of model checking C++ programs. In: *15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022*. IEEE, 2022. p. 461. Disponível em: <https://doi.org/10.1109/ICST53961.2022.00054>. 15, 26

MONTEIRO, F. R. et al. Bounded model checking of C++ programs based on the qt cross-platform framework. *Softw. Test. Verification Reliab.*, v. 27, n. 3, 2017. Disponível em: <https://doi.org/10.1002/stvr.1632>. 15

MYTKOWICZ, T. et al. Evaluating the accuracy of java profilers. In: . [S.l.: s.n.], 2010. v. 45, p. 187–197. 28

NONG, Y. et al. Evaluating and comparing memory error vulnerability detectors. *Information and Software Technology*, v. 137, p. 106614, 2021. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584921000896>. 28, 31

OLIVEIRA, V.; TEIXEIRA, L.; EBERT, F. On the adoption of kotlin on android development: A triangulation study. In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.: s.n.], 2020. p. 206–216. 15

Oracle. 2023. Acesso em: 16 de Fevereiro de 2023. Disponível em: <https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>. 15, 24

PATIL, T. et al. Comparative analysis of code optimization techniques with eminent applications (tools). 08 2022. 25

PEREIRA, P. A. et al. Verifying CUDA programs using smt-based context-bounded model checking. In: OSSOWSKI, S. (Ed.). *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*. ACM, 2016. p. 1648–1653. Disponível em: <https://doi.org/10.1145/2851613.2851830>. 15

RASHID, J.; MAHMOOD, T.; NISAR, M. A study on software metrics and its impact on software quality. 05 2019. 24

ROCHA, H. et al. Map2check: Using symbolic execution and fuzzing - (competition contribution). In: BIERE, A.; PARKER, D. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*. Springer, 2020. (Lecture Notes in Computer Science, v. 12079), p. 403–407. Disponível em: <https://doi.org/10.1007/978-3-030-45237-7_29>. 15

SAMENEZES, R. *jimple2json*. GitHub, 2023. Acesso em: 16 de Fevereiro de 2023. Disponível em: <https://github.com/rafaelsamenezes/jimple2json>. 38

SEMICONDUCTORS, N. *Techniques and Tools for Software Analysis*. [S.l.], 2011. Acesso em: 15 de Fevereiro de 2023. Disponível em: <https://www.nxp.com/docs/en/white-paper/CWTESTTECHCW.pdf>. 25, 26

SILVA, B. et al. Segurança de software em sistemas embarcados: Ataques defesas. In: ____. [S.l.: s.n.], 2013. p. 101 – 155. ISBN 978-85-7669-275-1. 15

SILVA, B. et al. Segurança de software em sistemas embarcados: Ataques defesas. In: ____. [S.l.: s.n.], 2013. p. 101 – 155. ISBN 978-85-7669-275-1. 23, 26, 31

SILVA, T. et al. *Verifying Security Vulnerabilities in Large Software Systems using Multi-Core k-Induction*. 2021. 26

SINGH, D. T. The hotspot java virtual machine: Memory and architecture. *International Journal of Allied Practice, Research and Review*, v. 1, p. 2350–1294, 10 2014. 24

TANG, D.; PLSEK, A.; VITEK, J. Memory safety for safety critical java. In: ____. [S.l.: s.n.], 2012. p. 235–254. ISBN 9781441981578. 23

TEAM, T. S. *Soot*. 2023. Acesso em: 18 de Fevereiro de 2023. Disponível em: <https://soot-oss.github.io/soot/>. 38

UNIDOS, C. de Segurança da Informação do Exército dos E. *CSI Software Memory Safety*. [S.l.], 2022. Disponível em: <https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF>. 16, 23

WHITTAKER, J. A.; THOMPSON, H. M. *How to break software security* /. Boston,: Pearson Addison Wesley,, 2003. 17

YOURKIT. *YourKit Java Profiler*. 2021. Acesso em: 18 de Fevereiro de 2023. Disponível em: <https://www.yourkit.com/java/profiler/>. 26

A

BENCHMARKS

Algoritmo A.1 – ListaDeObjetos.kt.

```
1  class ListaDeObjetos {
2      private val list = mutableListOf<Objeto>()
3
4      fun addObjeto() {
5          list.add(Objeto())
6      }
7  }
8
9  class Objeto {
10     val data = ByteArray(1024 * 1024)
11 }
12
13 fun main() {
14     val listaDeObjetos = ListaDeObjetos()
15
16     while (true) {
17         listaDeObjetos.addObjeto()
18         Thread.sleep(1000)
19     }
20 }
```

Algoritmo A.2 – ListaDeObjetosLimitada.kt.

```
1  class ListaDeObjetos {
2      private val list = mutableListOf<Objeto>()
3      private var count = 0
4      private val MAX_OBJETOS = 100
5
6      fun addObjeto() {
7          if (count >= MAX_OBJETOS) {
8          }
9              list.removeAt(0)
10         } else {
11             count++
12         }
13         list.add(Objeto())
14     }
15 }
16
17
18 class Objeto {
19     val data = ByteArray(1024 * 1024)
20 }
21
22 fun main() {
23     val listaDeObjetos = ListaDeObjetos()
24
25     while (true) {
26         listaDeObjetos.addObjeto()
27         Thread.sleep(1000)
28     }
29 }
```

Algoritmo A.3 – int_oflow.kt.

```
1  import java.util.Random;
```

```
2
3 fun intOverflow(): Int {
4     val A = Random().nextInt(30);
5     val B = Random().nextInt(30);
6     val sum = A + B
7     return sum
8 }
9 fun main() {
10    intOverflow()
11 }
```

Algoritmo A.4 – fatorial_overflow.kt.

```
1 fun fatorial(n: Int): Int {
2     return n * fatorial(n - 1)
3 }
4
5 fun main() {
6     val num = 5
7     val resultado = fatorial(num)
8 }
```

Algoritmo A.5 – array_oob.kt.

```
1 import java.util.Random;
2
3 fun main(){
4     var arr = IntArray(5)
5     var index = Random().nextInt(100)
6     arr[index] = 2
7 }
```

Algoritmo A.6 – StackOFlow.kt.

```
1 fun recursiveFunction(x: Int): Int {
2     return recursiveFunction(x + 1)
```

```
3 }
4
5 fun main() {
6     recursiveFunction(0)
7 }
```

Algoritmo A.7 – db0.kt.

```
1 fun main() {
2     val x = 1
3     val y = 0
4     val div = x/y
5 }
```