Verificação Formal de Software para Internet das Coisas com Validação de Aplicações em Kotlin/Java

Evelim B. Rocha, Vandermi J. da Silva, Andrey A. de O. Rodrigues

[1]Instituto de Ciências Exatas e Tecnologia – Universidade Federal do Amazonas (UFAM)

{evelim.rocha, vandermi, andreyrodrigues}@ufam.edu.br

Abstract. *With the increase in software within embedded systems, formal verification has become more important to ensure product quality by using mathematical models to identify unexpected behaviors during algorithm execution. Among the formal verification techniques, model checking stands out, as it verifies whether a system satisfies specific properties. This is essential for critical IoT systems developed in Java and, more recently, Kotlin.*
This research aims to apply formal verification to Android applications for IoT, identifying security and memory flaws, with the goal of evaluating the effectiveness of the tools and contributing to the improvement of these systems.

Resumo. *Com o aumento de softwares em sistemas embarcados, a verificação formal ganha importância para garantir a qualidade dos produtos, utilizando modelos matemáticos para identificar comportamentos imprevistos durante a execução de algoritmos.*
Dentre as técnicas de verificação formal, destaca-se o model checking, que verifica se um sistema atende a propriedades específicas, sendo essencial para sistemas críticos IoT desenvolvidas em Java e, mais recentemente, Kotlin.
Esta pesquisa visa aplicar verificações formais em aplicações Android para IoT, identificando falhas de segurança e memória, com o objetivo de avaliar a eficácia das ferramentas e contribuir para a melhoria desses sistemas.

1. Introdução

O uso de dispositivos com capacidade computacional e acesso à Internet tornou-se cada vez mais comum no cotidiano das pessoas [Rowland et al. 2015]. Com o avanço da tecnologia, a Internet das Coisas (do inglês *Internet of Things* — IoT) tem contribuído significativamente para o aumento da conectividade e da automação em diversos setores, como o industrial e o tecnológico. Esse avanço promove o crescimento contínuo no uso de dispositivos inteligentes interconectados [Moraes and Hayashi 2021].

Paralelamente, observa-se um aumento expressivo na quantidade de software embarcado em tais dispositivos. Esse cenário exige mecanismos eficazes de verificação, sendo a verificação formal uma alternativa promissora para assegurar a confiabilidade e a qualidade do produto [SILVA et al. 2013]. Embora existam outros métodos de verificação amplamente utilizados, a verificação formal se destaca por permitir a análise de comportamentos não determinísticos por meio de modelos matemáticos, possibilitando a detecção de comportamentos imprevistos durante a fase de testes [Ramalho et al. 2013].

Nesse contexto, esta pesquisa visa explorar a verificação formal de aplicações Android voltadas à IoT, desenvolvidas em Kotlin/Java. O objetivo é identificar vulnerabilidades relacionadas à segurança e à memória, como *deadlock*, *memory leak*, *arithmetic overflow* e *data race*, utilizando ferramentas compatíveis com essas linguagens.

2. Objetivos

2.1. Objetivo Geral

Aplicar as técnicas de verificação formal para aplicações desenvolvidas com o intuito de identificar *deadlock*, *memory leak*, *arithmetic overflow* e *data race* em *bytecodes* gerados pelas linguagens Kotlin/Java.

2.2. Objetivos Específicos

- Realizar um levantamento bibliográfico sobre as ferramentas de verificação formal que são utilizadas em bytecode Kotlin/Java.
- Desenvolver testes com verificação e inspeção de *bytecodes*.
- Identificar *deadlock*, *memory leak*, *arithmetic overflow* e *data race* em *bytecode* Kotlin/Java.

3. Fundamentação Teórica

A verificação de modelos (*model checking*) é uma técnica de verificação formal amplamente utilizada para garantir que um sistema atenda a propriedades previamente especificadas. Diferentemente dos testes convencionais, que utilizam entradas determinísticas, o *model checking* permite uma análise abrangente, explorando todas as possíveis execuções do sistema, inclusive as não determinísticas, o que é essencial em sistemas críticos [Clarke 1997].

Essa abordagem exige a formulação de especificações formais que descrevem propriedades desejáveis, como ausência de *deadlocks*, gerenciamento correto de *buffers* e integridade de dados. Assim, torna-se possível identificar falhas complexas como *arithmetic overflow*, *deadlock*, *data race* e *memory leak*, que podem passar despercebidas em abordagens tradicionais.

Deadlocks ocorrem quando dois ou mais processos entram em espera circular por recursos mantidos entre si, impedindo a continuação da execução. Um caso simples envolve dois processos, cada um mantendo um recurso e aguardando pelo recurso do outro, resultando em bloqueio mútuo [Isloor and Marsland 1980].

Memory leaks surgem quando objetos que não são mais necessários permanecem acessíveis por referências estáticas ou locais, geralmente devido a descuidos do programador, como esquecer de remover objetos de estruturas de dados. Esses objetos não podem ser coletados automaticamente, levando à degradação de desempenho ao longo do tempo [Weninger et al. 2019].

O *arithmetic overflow* ocorre quando um valor excede os limites representáveis pelo tipo de dado utilizado. Por exemplo, somar 1 ao valor 127 em um inteiro com sinal de 8 bits (0x7F) resulta em 0x80, que representa -128 em vez de +128, configurando uma violação de integridade numérica [Mihocka and Troeger 2010].

A condição de *data race* se manifesta quando dois ou mais threads acessam simultaneamente uma variável compartilhada e ao menos uma das operações é de escrita, sem sincronização adequada. Esse tipo de erro leva a resultados imprevisíveis e comportamentos incorretos na execução [Pozniansky and Schuster 2003].

Para lidar com esses desafios, técnicas como o *Bounded Model Checking* (BMC) vêm sendo amplamente utilizadas. O BMC verifica a violação de propriedades em profundidades limitadas de execução, possibilitando a análise de sistemas complexos de forma decidível [Biere et al. 2009]. Embora eficaz, ainda há a necessidade de evolução nas ferramentas que implementam essa técnica [Monteiro et al. 2022].

Algumas ferramentas, como JBMC [Cordeiro et al. 2018] e JayHorn [Kahsai et al. 2019], oferecem suporte à verificação de *bytecode* Java, porém enfrentam limitações ao tratar programas escritos em Kotlin. Para suprir essa lacuna, os autores de Menezes propuseram uma abordagem que combina o BMC, por meio da ferramenta ESBMC, com a linguagem intermediária Jimple e técnicas de *fuzzing*, com foco em detectar falhas como estouros de memória e vulnerabilidades de concorrência, especialmente em implementações criptográficas de IoT.

Os autores destacam que, apesar de várias ferramentas oferecerem suporte ao Java, apenas JBMC e ESBMC-Jimple mostraram compatibilidade efetiva com Kotlin. O estudo comparativo conduzido por Menezes revelou que o ESBMC-Jimple obteve desempenho superior na verificação de *benchmarks* em Kotlin, em relação ao JBMC. Essa vantagem é atribuída à utilização de comandos nativos e à adaptação da linguagem intermediária Jimple.

A técnica empregada no JBMC baseia-se em um *frontend* especializado na análise de *bytecode*, com suporte à simulação de chamadas de métodos virtuais e à detecção de falhas não tratadas pela JVM [Cordeiro et al. 2018]. Já o ESBMC-Jimple realiza análise estática diretamente sobre representações em Jimple, permitindo maior precisão na identificação de vulnerabilidades [Menezes et al. 2022].

Diante desses resultados, o ESBMC-Jimple foi selecionado como ferramenta principal para a análise experimental deste projeto, visando avaliar sua eficácia na detecção de falhas em aplicações reais. A escolha fundamenta-se na necessidade de garantir a segurança e a robustez dos sistemas IoT, cuja criticidade cresce exponencialmente nos cenários atuais.

4. Metodologia

As três etapas que compõem a metodologia são: mapeamento sistemático (MSL), análise da linguagem Kotlin e inspeção formal de *bytecodes*.

- **Mapeamento Sistemático da Literatura (MSL):** Essa etapa é crucial para identificar as limitações dos métodos existentes e sugerir possíveis melhorias. Para isso, foi realizada a busca por artigos científicos da área da pesquisa nas bases de dados da IEEE, Scopus e ACM para realizar o MSL da área de pesquisa.
- **Análise da Linguagem Kotlin:** Para realizar a análise, foi realizado um estudo com codificação de funcionalidades em Kotlin para simular *deadlock*, *memory leak*, *arithmetic overflow* e *data race*, com objetivo de entender o uso das ferramentas existentes para verificação formal dessas linguagens e avaliar os protótipos.

- **Inspeção Formal de *bytecodes*:** Nessa etapa, foram desenvolvidas aplicações utilizando as linguagens estudadas nas quais serão aplicadas as técnicas de verificação formal, analisar os resultados e sugerir melhorias na codificação. Os resultados serão comparados com os encontrados na literatura.

Durante a realização da pesquisa, foram utilizados exemplos de código em Kotlin executados no ESBMC-Jimple com o objetivo de identificar problemas relacionados a propriedades de execução, como impasses, exceções não tratadas e possíveis caminhos de execução indesejados.

5. Resultados e Discussão

O estudo conduzido pelos autores de ESBMC-Jimple [Menezes et al. 2022] apresenta uma análise comparativa de verificação em linguagem Kotlin, através de uma avaliação experimental. Os resultados indicaram que a ferramenta demonstrou uma capacidade superior em verificar corretamente um maior número de *benchmarks* Kotlin em comparação com o JBMC. Essa superioridade pode ser atribuída aos comandos nativos do ESBMC-Jimple, que são eficazes na detecção de problemas específicos relacionados à questão secundária abordada.

Em virtude dessas descobertas, o ESBMC-Jimple foi selecionado como o único verificador para avaliar sua eficácia na detecção de vulnerabilidades em aplicações reais, como sistemas IoT, nos testes realizados neste projeto. A escolha se justifica pela necessidade de garantir a segurança e a integridade de tais sistemas, que são cada vez mais críticos em diversos contextos tecnológicos.

5.1. Amostragem de Resultados da Pesquisa

Para a realização final da amostragem de resultados, foram desenvolvidos quatro simulações de experimento para simular os problemas de segurança e memória: *deadlock*, *memory leak*, *arithmetic overflow* e *data race*. Todas as etapas da amostragem e detecção de problemas com o verificador estão detalhados nos seguintes tópicos.

5.1.1. Setup de Experimento

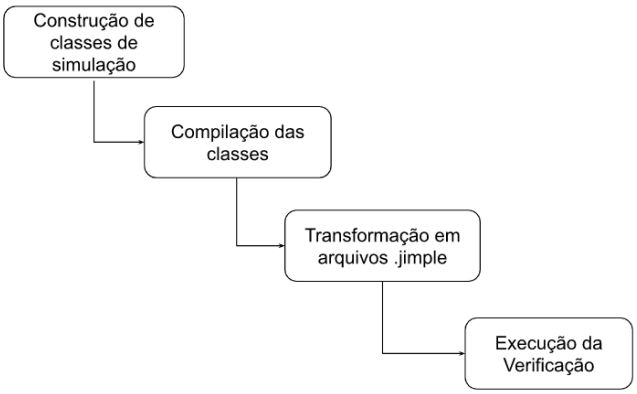
O ambiente de desenvolvimento utilizado foi o sistema operacional Linux Ubuntu 20.04 com 8Gb de RAM. A versão utilizada da ferramenta foi a segunda versão disponível e utilizada no trabalho de Menezes, encontrada juntamente com seus benchmarks em https://zenodo.org/records/6514608. Para utilizá-la, necessita ter Java e Kotlin instalado em sua máquina. Os autores dessa versão realizaram uma demonstração real de como a ferramenta funciona, encontrada em https://youtu.be/J6WhNfXvJNc.

5.1.2. Sequência de Passos

Terminada a instalação e configuração do ESBMC-Jimple, foi utilizada a seguinte sequência de passos no experimento:

A sequência de passos consiste em fazer a construção das classes de simulação, onde irá ocorrer a verificação de possíveis violações nas estruturas de dados. O próximo

Figura 1. Sequência de passos utilizados no experimento



passo é a compilação dos códigos kotlin utilizando o arquivo nativo da ferramenta "compile-kotlin.sh". Em seguida, as classes são transformadas em arquivos ".jimple" usando o arquivo "generate-jimple.sh" com seus arquivos .class. Por fim, para demonstrar o resultado da verificação, há a execução usando o "esbmc" juntamente com seus arquivos .jimple e os comandos desejados para a análise de *bytecode*.

5.1.3. Amostragem do Experimento

No processo de verificação, a ferramenta gera alguns arquivos extra para auxiliar na execução. Devido a isso, as classes foram separadas em diferentes diretórios para melhor visualização e clareza de seus passos. A verificação das classes foi realizada de forma separada, onde cada uma de suas classes tinha sua própria função "main" como indica o exemplo no *listing* 1.

Listing 1. Classe teste "main.kt"que simula um overflow

```
1   class OverFlowSimulate {
2       var a = 20000000L
3       var b = 20000000L
4       var c = 20000000L
5
6       var d = a + b + c
7       var e = d * d
8
9       fun imprimeNumero() {
10          println("e=$e")
11      }
12  }
13
14  fun main() {
15      val overFlow = OverFlowSimulate()
16      overFlow.imprimeNumero()
17  }
```

No processo inicial da coleta de resultados, houveram desafios na integração do

código Kotlin para com ESBMC-Jimple. Após a compilação, as classes, principalmente de *deadlock* e *memory simulate*, não aceitavam a conversão para a extensão .jimple utilizando o SOOT, uma estrutura de análise estática para *bytecode* Java, com código Kotlin. A compatibilidade e a conversão adequada do código Kotlin em um formato que pudesse ser processado pelo SOOT foram problemas iniciais que surgiram.

Apesar desses conflitos, foi encontrada outra solução para o problema. No repositório do autor dos *scripts* do ESBMC-Jimple, é disponibilizada uma alternativa para a conversão de *bytecodes* Kotlin para Jimple. Essa alternativa está disponível no seguinte repositório GitHub.

Após a conversão de *bytecodes*, a verificação foi realizada no prompt de comando com a seguinte instrução do *listing* 2

Listing 2. Instrução para a verificação com a classe OverFlowSimulate

```
/home/usuario/ESBMC-Jimple/esbmc sootOutput/mainKt.jimple
    sootOutput/OverFlowSimulate.jimple --overflow-check --k-
    induction
```

Listing 3. Resultado da verificação no terminal

```
Violated proprerty:
  aritmethic overflow on add
  !overflow("*", $16, $15)
VERIFICATION FAILED
```

A verificação estendeu-se para as demais classes DataRaceCondition, DeadlockSimulate e MemoryLeakSimulate, resultando na tabela 1. A ferramenta possui verificadores de propriedades nativas para cada tipo de erro especificado, facilitando, assim, a detecção de problemas próprio de cada classe testada.

Tabela 1. Resultado das verificações com ESBMC-Jimple

Classes	Verificador de Propriedade	Erro	LC	T (s)	M (bytes)	Tipo de Erro (TE)
DataRaceCondition	–data-races-check	Sim	48	0,59	16252928	Corrida de Dados
DeadlockSimulate	–deadlock-check	Sim	46	0,63	16252928	Bloqueio Fatal
MemoryLeakSimulate	–memory-leak-check	Sim	21	0,61	49913856	Vazamento de Memória
OverFlowSimulate	–overflow-check	Sim	17	0,14	16252928	Estouro Aritmético

Na tabela 1, a primeira coluna possui o nome das classes seguida do verificador de propriedade utilizado na verificação, já a terceira coluna apresenta se o erro foi encontrado. As colunas seguintes estão com o título simplificado em siglas onde LC representa quantas linhas de código a classe possui, T (s) é o tempo de execução, em segundos, da classe pelo ESBMC-Jimple, M (bytes) é a quantidade de memória RAM quantificada em bytes e TE é o tipo do erro encontrado traduzido para o português.

É observável que os erros influenciaram na quantidade de tempo e memória utilizada pelo hardware, podendo destacar a classe DeadlockSimulate com o maior tempo de execução e MemoryLeakSimulate pela maior quantidade de memória utilizada. Vale mencionar que a primeira classe dita, se executada normalmente pelo compilador Kotlin, terá tempo de execução infinito, ou seja, o programa nunca terminará o processo impossibilitando o funcionamento da aplicação.

6. Considerações Finais

Essa pesquisa aplicou técnicas de verificação formal em aplicações com o intuito de identificar *deadlock*, *memory leak*, *arithmetic overflow* e *data race* em *bytecodes* gerados pelas linguagens Kotlin/Java. Esse objetivo foi alcançado, como descrito no tópico anterior, onde foram desenvolvidas aplicações-teste para a ferramenta ESBMC-Jimple.

Embora a verificação tenha sido realizada sem maiores problemas, há uma notável dificuldade para com o uso da ferramenta, no qual necessita de manipulação via terminal para configurar e executar os testes. É uma limitação para usuários inexperientes em verificadores formais e demanda de orientações externas de instalação de dependências, conversão para extensão .jimple e comandos não especificados no repositório existente.

Apesar da complexidade, o verificador estudado mostrou-se eficaz quanto à identificação de vulnerabilidades. A verificação revelou que a ferramenta consegue identificar com precisão todas as vulnerabilidades listadas no objetivo da pesquisa, além de indicar a localização da causa dos erros, o que auxilia na refatoração pelo desenvolvedor.

Conclui-se que, baseado nos dados experimentais disponíveis até o momento, a ferramenta cumpre sua função de verificação formal ao identificar os problemas especificados com alta assertividade, mostrando-se ideal para a realização de verificações minuciosas em aplicações Kotlin. No contexto de IoT, a ferramenta pode auxiliar na prevenção de erros de vulnerabilidade de hardware aos componentes interconectados do sistema de Internet das Coisas.

Referências

[Biere et al. 2009] Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., and Zhu, Y. (2009). Bounded model checking. *Handbook of satisfiability*, 185(99):457–481.

[Clarke 1997] Clarke, E. M. (1997). Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer.

[Cordeiro et al. 2018] Cordeiro, L., Kesseli, P., Kroening, D., Schrammel, P., and Trtik, M. (2018). Jbmc: A bounded model checking tool for verifying java bytecode. In *International Conference on Computer Aided Verification*, pages 183–190. Springer.

[Isloor and Marsland 1980] Isloor, S. S. and Marsland, T. A. (1980). The deadlock problem: An overview. *Computer*, 13(9):58–78.

[Kahsai et al. 2019] Kahsai, T., Rümmer, P., and Schäf, M. (2019). Jayhorn: A java model checker: (competition contribution). In *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 25*, pages 214–218. Springer.

[Menezes et al. 2022] Menezes, R., Moura, D., Cavalcante, H., de Freitas, R., and Cordeiro, L. C. (2022). Esbmc-jimple: verifying kotlin programs via jimple intermediate representation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 777–780.

[Mihocka and Troeger 2010] Mihocka, D. and Troeger, J. (2010). A proposal for hardware-assisted arithmetic overflow detection for array and bitfield operations. In *WISH—Workshop on Infrastructures for Software/Hardware Co-Design*. Citeseer.

[Monteiro et al. 2022] Monteiro, F. R., Gadelha, M. R., and Cordeiro, L. C. (2022). Model checking c++ programs. *Software Testing, Verification and Reliability*, 32(1):e1793.

[Moraes and Hayashi 2021] Moraes, A. d. and Hayashi, V. (2021). *Segurança em IoT: Entendendo os Riscos e Ameaças em IoT. Alta Books*. Editora Alta Books.

[Pozniansky and Schuster 2003] Pozniansky, E. and Schuster, A. (2003). Efficient on-the-fly data race detection in multithreaded c++ programs. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190.

[Ramalho et al. 2013] Ramalho, M., Freitas, M., Sousa, F., Marques, H., Cordeiro, L., and Fischer, B. (2013). Smt-based bounded model checking of c++ programs. In *2013 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*, pages 147–156. IEEE.

[Rowland et al. 2015] Rowland, C., Goodman, E., Charlier, M., Light, A., and Lui, A. (2015). *Designing connected products: UX for the consumer Internet of Things*. "O'Reilly Media, Inc.".

[SILVA et al. 2013] SILVA, V. J., CORDEIRO, L. C., and JÚNIOR, V. F. D. L. (2013). Verificação de aplicações ami usando java pathfinder.

[Weninger et al. 2019] Weninger, M., Gander, E., and Mössenböck, H. (2019). Analyzing data structure growth over time to facilitate memory leak detection. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 273–284.