Actas do décimo quarto Simpósio de Informática (INForum 2023)

Análise do Consumo de Energia de Aplicações Android

Paul Robert[1,2]*, Délcio Ferramenta[2]†, and Simão Melo de Sousa[2]

[1] Ecole Normale Supérieure Paris Saclay, France
[2] NOVA-LINCS, Release Lab, DI-FE, Universidade da Beira Interior, Portugal

Resumo

O uso de energia em dispositivos portáteis é um elemento de extrema importância para a satisfação do usuário e o desempenho eficiente dos dispositivos. A crescente busca por aplicativos e a abundância de escolhas nas lojas são uma realidade cada vez mais presente na vida das pessoas. No entanto, é essencial que os usuários tenham consciência dos efeitos energéticos das aplicações instaladas em seus dispositivos. Este artigo apresenta uma abordagem estática para estimar o consumo de energia em aplicativos Android. O objetivo é estabelecer uma métrica que permita uma avaliação justa e comparativa entre os aplicativos, tornando possível classificá-los de forma adequada. Para alcançar esse objetivo, utilizamos a métrica chamada *Worst Case Execution Cost* (WCEC), que representa a quantidade máxima de energia consumida durante a execução mais intensiva de um programa. Para estimar o WCEC, propomos uma definição precisa do consumo de energia das instruções Android, baseada em um modelo de custo. Além disso, empregamos métodos de geração de problemas *ILP (Integer Linear Programming)* para modelar uma aproximação desse consumo. Aplicamos também técnicas de interpretação abstrata para refinar as restrições *ILP* geradas, a fim de obter estimativas mais precisas. Essas ideias são implementadas em um protótipo, o qual será brevemente apresentado para demonstrar a eficácia de nossa abordagem.

Palavras-chave: Análise estática, WCEC, bytecode, Android, modelo energético.

1 Introdução

O projeto *GreenStamp - Mobile Energy Efficiency Services*, é um projeto de pesquisa industrial que busca avaliar o consumo de energia de aplicativos Android. O principal objetivo desse projeto é fornecer informações valiosas as lojas de aplicativos, para poder classificar os aplicativos de acordo o seu consumo energético. Dessa forma, os usuários poderão fazer escolhas mais conscientes em relação ao gasto energético dos aplicativos que desejam utilizar. No projeto GreenStamp foram desenvolvidas diversas abordagens, que podem ser categorizadas em análises estáticas e análises dinâmicas.

As análises estáticas concentram-se em examinar o consumo de energia de um aplicativo em um estado fixo, sem levar em consideração as variações ao longo do tempo. Essas análises fornecem uma visão geral do consumo de energia. Por outro lado, as análises dinâmicas estão voltadas para a compreensão do consumo de energia em tempo real, levando em consideração apenas as variações durante a execução do aplicativo. Ao combinar essas duas abordagens, é possível obter uma visão mais abrangente do comportamento geral do aplicativo.

O parceiro industrial desse projeto gerencia uma das maiores lojas de aplicativos Android do mundo.

*O trabalho foi apoiado pela **NOVA-LINCS** (UIDB/04516/2020) com o apoio financeiro da FCT Fundação para a Ciência e a Tecnologia, através de fundos nacionais.

†O trabalho foi financiado pelo FEDER (Fundo Europeu de Desenvolvimento Regional), da União Europeia, através do CENTRO 2020 (Programa Operacional Regional do Centro), no âmbito do projeto CENTRO-01-0247-FEDER-047256 - **GreenStamp**: Serviços de Eficiência Energética para Dispositivos Móveis.

Neste contexto, compreende-se que o consumo de energia em dispositivos móveis é um fator de grande relevância para a experiência do usuário e a eficiência operacional dos dispositivos. A crescente demanda por aplicativos e a diversidade de opções disponíveis nas lojas tornam-se uma realidade cada vez mais presente no cotidiano dos usuários. No entanto, é crucial que os utilizadores estejam cientes dos impactos energéticos das aplicações instaladas nos seus dispositivos.

O objetivo desta pesquisa é desenvolver uma análise estática capaz de gerar uma métrica para comparar o consumo de energia de aplicativos Android. Uma abordagem natural para estimar o consumo energético de programas, usando a análise estática, é utilizar o conceito de *Worst Case Energy Consumption* (WCEC - Consumo de Energia no Pior Caso). O WCEC representa a quantidade máxima de energia consumida em um cenário de utilização mais intensiva de um programa ou sistema.

A análise de WCEC é baseada na análise de *Worst Case Execution Time* (WCET - Tempo de Execução no Pior Caso). Como a análise de WCET é mais bem estabelecida em comparação com a análise de WCEC, realizamos um estudo aprofundado sobre ela para embasar nossas abordagens. Nas próximas seções, apresentaremos uma contextualização detalhada dessas análise.

Para a nossa analise do WCEC para aplicativos moveis estabelecemos os seguintes critérios:

1. A análise deve ser estática: não será necessária a execução das aplicações alvo;

2. O objetivo é obter critérios de comparação entre os aplicativos: o resultado da análise não precisa ser absolutamente preciso;

3. Na medida do possível, a análise deve ser independente do hardware e fácil de configurar para um hardware específico.

2 Estado da arte

A análise WCET é estudada há mais de 20 anos. Uma apresentação do problema e suas soluções é fornecida em [1]. A abordagem dinâmica usual envolve a execução do programa várias vezes com diferentes parâmetros para determinar o pior tempo de execução na prática. Essa abordagem não fornece garantia sobre os resultados e apresenta riscos para sistemas críticos onde a precisão temporal é importante, ou até mesmo vital.

Como alternativa, foram desenvolvidas abordagens estáticas para abordar o WCET. Nessa abordagem, o problema é formulado como um Programa Linear Inteiro (ILP - *Integer Linear Programming*), criado a partir das restrições de fluxo e tempo [2]. Esses métodos são frequentemente aplicados a códigos controlados (por exemplo, o limite dos ciclos são explicitadas pelo programador), o que permite gerar mais restrições e obter resultados mais precisos do que a simples resolução de um ILP ou considerando apenas os caminhos de execução possíveis. Além disso, é possível modelar o comportamento da *cache* e do pipeline usando técnicas de interpretação abstrata [1, 3]. A precisão e validade do tempo de execução atribuído a cada instrução do processador também são cruciais para obter um WCET correto. Essas suposições fortes (conhecimento preciso do hardware de execução e precisão explícita dos caminhos de execução) são razoáveis em contextos industriais que usam programas críticos em processadores bem conhecidos. No entanto, elas são completamente irrealistas para o *bytecode* Android, que é portável e possui centenas de milhares de linhas de código geradas automaticamente. Portanto, os trabalhos apresentados aqui permanecerão simplificados em sua abordagem.

O problema do WCEC é o análogo do WCET para energia, em vez de tempo. Os trabalhos sobre esse problema são mais recentes e menos numerosos [4, 5], embora utilizem técnicas semelhantes. Além disso, é bem estabelecido [6] que não há correlação direta entre WCET e WCEC: modelos de tempo e energia específicos para os respetivos problemas devem ser usados.

No contexto do *bytecode* Android e sua formalização, existem muitos trabalhos que se baseiam principalmente na pesquisa em torno do *bytecode* Java. Foram desenvolvidos métodos de execução simbólica [7], assim como a formalização semântica do *bytecode* [8]. No entanto, nenhum desses trabalhos leva em consideração as otimizações mais recentes do ambiente Android. Também existem ferramentas de análise estática [9], que geralmente reutilizam as ferramentas Java existentes. A mais famosa é o *Soot* [10], que se concentra na análise do fluxo de dados e usa uma representação interna chamada `Jimple` [11], obtida por descompilação do *bytecode* Android. O uso dessa representação intermediária complicaria as análises WCET/WCEC, que normalmente são realizadas no código de nível mais baixo possível, e é a principal razão para não usar o *Soot* neste projeto.

3 Máquina Virtual Android

A Máquina Virtual Android é uma parte fundamental do sistema operacional Android e é responsável por executar os aplicativos desenvolvidos para essa plataforma. Embora seja derivada da máquina virtual Java, a Máquina Virtual Android apresenta algumas diferenças importantes para melhor se adequar aos requisitos específicos do sistema Android.

A versão atual da máquina virtual, ART (*Android Runtime*), é altamente otimizada e utiliza técnicas como compilação *"Ahead Of Time"* (AOT) e *"Just In Time"* (JIT), para melhorar o desempenho dos aplicativos. Essas otimizações envolvem recompilar o aplicativo regularmente com base em perfis de uso durante instalação e execução. No entanto, é importante ressaltar que a análise apresentada neste trabalho considera uma interpretação simples do bytecode, sem levar em conta essas otimizações avançadas. Dessa forma, a análise é realizada diretamente sobre o bytecode original dos aplicativos, sem considerar as recompilações e otimizações específicas realizadas pela Máquina Virtual Android.

Embora essa simplificação possa criar uma distância entre o cenário considerado pelo analisador e uma execução real mais complexa e otimizada, a análise proposta ainda é válida e razoável. Ela se concentra nas instruções mais relevantes em termos de tempo de execução e consumo de energia, como os métodos nativos e pré-compilados que têm um impacto significativo no desempenho do aplicativo.

O *bytecode* de um aplicativo é contido em vários arquivos *.dex*, que podem ser descompilados para leitura. Cada arquivo *.dex* contém uma coleção de classes, campos e métodos. Cada método contém um corpo de *bytecode*, que consiste em 224 instruções semelhantes a um *assembly* do tipo x86, incluindo instruções como `goto` e `jmp`. A máquina virtual utiliza 256 registores capazes de armazenar diferentes tipos de valores, incluindo objetos e matrizes. É importante mencionar que, devido à complexidade de lidar com a presença de instruções de *multithreading* de forma estática, essa característica será ignorada neste contexto. Os métodos *invoke* são utilizados para chamar outros métodos, que podem ser definidos no próprio *bytecode*, ser nativos do sistema Android ou pré-compilados em arquivos APK.

4 Modelo de Custo

A complexidade do sistema Android torna difícil obter propriedades estáticas precisas de um programa Android, como o consumo de energia e o tempo de execução, devido às otimizações específicas realizadas pela Máquina Virtual Android, como a compilação AOT e JIT. No entanto, a análise se concentra nos métodos não compilados encontrados no arquivo *.dex* que contém o *bytecode*. Mesmo com essa simplificação, é possível obter informações relevantes sobre o fluxo de execução do programa e a natureza dos métodos nativos invocados.

Neste contexto, assumiremos que esses métodos nativos são conhecidos e que possuímos um modelo de custo em termos de tempo e energia para cada um dos métodos nativos, que pode ser usado para uma análise de consumo máximo de energia/tempo (WCEC/WCET) da aplicação.

Em um sentido amplo, um modelo de custo energético é uma função c que associa cada instrução i do conjunto de *bytecode* I a um elemento de um grupo $(E, 0_E, +_E)$ representando o consumo de energia da execução dessa instrução. Apresentamos essa representação como um número de ponto flutuante, que é uma das formas mais simples de representação. O consumo total de energia de uma sequência de instruções será a soma dos consumos de energia de cada instrução individualmente.

Construção do modelo de custo. Para estabelecer um modelo de custo de referência no nível das instruções Android, escolhemos um modelo de smartphone fixo que sirva como referência. O smartphone XIAOMI Redmi Note 9 Pro (XRN9) foi selecionado como alvo para essa análise, pois sua arquitetura de hardware e sistema são representativos dos smartphones atuais.

Para reduzir o ruído potencial durante a medição do consumo de energia no nível das instruções, ativamos o modo de economia máxima de bateria, que tem como efeito, por exemplo, reduzir ao máximo o brilho da tela, desativar o cartão SIM para evitar a interferência dos serviços de telefonia móvel e garantir que não haja nenhum processo em segundo plano ativo.

É importante ressaltar que obter um modelo de custo preciso é desafiador, inclusive para os modelos de consumo de energia fornecidos pelos fabricantes de hardware, que podem não ser totalmente confiáveis [5]. Optamos por caracterizar o consumo de energia das instruções em relação umas às outras, conforme recomendado por [4]. Esses modelos de energia são chamados de modelos *relativos*. Para obter o modelo de custo para o smartphone de referência, criamos um conjunto de programas Android que repetem uma mesma instrução ou chamada a um método nativo um certo número de vezes. Durante a execução repetida desses programas, medimos precisamente o consumo instantâneo em intervalos regulares de tempo, garantindo as mesmas condições para todas as instruções analisadas.

Critério de consumo. O critério de consumo adotado para comparar aplicativos Android é baseado na ideia de buscar "picos"de consumo em um determinado período de tempo $t_{\max}$. Essa abordagem de buscar "picos"de consumo é uma adaptação necessária para lidar com a complexidade e a natureza não linear dos aplicativos Android. Diferente dos programas com garantia de terminação e com um único ponto de entrada, os aplicativos Android são compostos por múltiplos métodos que comumente podem executar de forma não determinística, o que inviabiliza a aplicação direta de métodos convencionais de análise do WCEC.

O tempo limite $t_{\max}$ representa um cenário realista em que os usuários interagem com o aplicativo. A análise focada nesse intervalo de tempo fornece informações relevantes sobre o desempenho do aplicativo durante a interação do usuário, permitindo identificar os trechos de código que são mais críticos em termos de consumo de energia/tempo nesse contexto.

5 Fluxo de funcionamento da Ferramenta

A nossa ferramenta de análise de WCEC dos aplicativos Android segue o fluxo apresentado na figura 1. A seguir, descrevemos o cada uma das etapas do fluxo:

1. **Entrada do Aplicativo (APK)**: A ferramenta tem como entrada o arquivo APK do aplicativo que se deseja analisar. O APK contém todas as informações e recursos do aplicativo, incluindo o bytecode compactado do aplicativo;

2. **Extração dos Arquivos .dex**: A ferramenta extrai os arquivos DEX contidos no APK. Cada arquivo .dex corresponde a um módulo do aplicativo contendo as classes e métodos compilados em bytecode;

3. **Decompilação do Bytecode**: A partir dos arquivos .dex, a ferramenta decompila o bytecode do aplicativo para obter uma representação em linguagem de alto nível mais próxima do código-fonte original do aplicativo.

4. **Construção do Grafo de Fluxo de Controle Interprocedural**: Com o código-fonte decompilado, a ferramenta constrói o grafo de fluxo de controle interprocedural que representa o fluxo de execução de toda a aplicação. Esse grafo contém informações sobre os métodos, chamadas de método, condições e loops presentes no código do aplicativo.

5. **Geração do Problema de ILP para cada Método**: Para cada método do aplicativo, a ferramenta formula um problema de Programação Linear Inteira (ILP) para calcular o WCEC específico daquele método. O ILP leva em conta as estimativas de consumo de energia/tempo das instruções, bem como as restrições obtidas pela interpretação abstrata.

6. **Refinamento do ILP**: Com o problema de ILP formulado, a ferramenta realiza um processo de refinamento para levar em conta a execução das instruções em diferentes caminhos do grafo de fluxo de controle. Essa etapa é importante para obter resultados mais precisos, pois leva em consideração a execução condicional e repetitiva do código.

7. **Interpretação Abstrata para Obtenção de Restrições**: Nesta etapa, a ferramenta realiza uma analise via interpretação abstrata do código-fonte para obter informações sobre os possíveis valores das variáveis e expressões em cada ponto do programa. Essas informações são usadas para gerar restrições para as variáveis no contexto do Programação Linear Inteira (ILP).

8. **Cálculo do WCEC para cada Método**: O ILP refinado é resolvido para cada método individualmente, resultando no WCEC estimado para aquele método em particular.

9. **Cálculo do WCEC Total**: Com os WCECs estimados para cada método, a ferramenta calcula o WCEC total do aplicativo, que representa o consumo máximo de energia/tempo para toda a aplicação.

Em resumo, o processo de análise envolve a extração e descompilação do bytecode, a construção e um ICFG (Interprocedural control Flow Graph - grafo de controle de fluxo interprocedimento) para representar o fluxo de controle, a obtenção de restrições por meio da interpretação abstrata, a formulação e refinamento dos problemas de ILP para cada método, e, finalmente, o cálculo do WCEC total da aplicação.

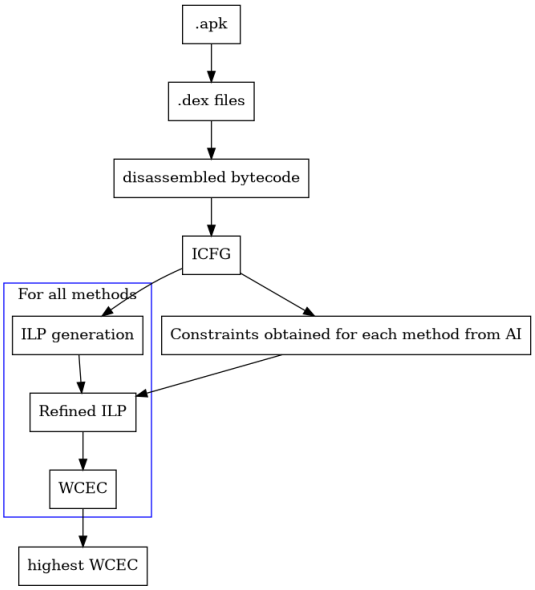


Figura 1: Fluxo de funcionamento da Ferramenta.

6 Gerando os ILPs

Nesta seção, apresentamos o processo de geração dos Problemas de Programação Linear Inteira (ILPs) e como eles são utilizados para calcular o WCEC de cada método do aplicativo e, em seguida, o WCEC total da aplicação.

6.1 Construção do CFG

Um *bytecode* Android contém o código de cada um de seus métodos declarados, os quais podem todos ser traduzidos em um CFG. Métodos podem invocar outros métodos por meio de instruções *invoke*. O trabalho de [12] detalha o processo de construção de um CFG Interprocedural (ICFG) para *bytecode* Java. Essas técnicas podem ser reaproveitadas para o Android.

Neste contexto, podemos conceituar o CFG como uma representação composta por um conjunto de blocos de instruções V, no qual cada bloco é uma sequência de comandos executados em ordem. O fluxo de controle entre esses blocos é definido por um conjunto de arestas direcionadas $E \subseteq V \times V$. Além disso, o método possui um bloco de entrada $s \in V$ e um conjunto de blocos de saída $R \subseteq V$.

Para simplificar o CFG, consideramos que uma instrução *invoke* representa um bloco de tamanho unitário. Além disso, as instruções de ramificação são sempre as últimas instruções em seus respectivos blocos e apontam para os blocos aos quais devem ser direcionados de acordo com a execução do programa. Essa definição possibilita a transformação de cada método em seu CFG.

Definimos o ICFG (Grafo de Controle de Fluxo Interprocedural) para um conjunto de CFGs, representado como $C = G_1, \cdots, G_n$. O ICFG é um tipo especial de grafo onde métodos podem se chamar mutuamente por meio de instruções *invoke*. É importante destacar que os argumentos desses *invoke* podem não corresponder a métodos cujos CFGs sejam elementos de C.Quando

isso ocorre, tais argumentos são simplesmente ignorados durante o processo de construção do grafo.

No entanto, nos casos em que um bloco *invoke* b_{inv} aponta para um bloco b'_{inv} que invoca um método associado ao CFG G_i, com bloco de entrada s_i e blocos de saída $r_1, \cdots r_m$, realizamos um conjunto de ações específicas:

- Removemos a aresta $b_{inv} \rightarrow b'_{inv}$;
- Adicionamos as arestas $b_{inv} \rightarrow s_i$, $r_1 \rightarrow b_{inv,r}, \cdots r_m \rightarrow b_{inv,r}$ e $b_{inv,r} \rightarrow b'inv$, sendo que $binv, r$ é um bloco vazio.

Essas operações visam estabelecer a devida conexão entre os blocos de instruções durante a construção do ICFG.

6.2 Procura dos Blocos Alcançáveis

Para determinar os blocos alcançáveis em $t_{\max}$ e estabelecer o WCEC, é necessário encontrar os blocos que podem ser alcançados dentro desse limite de tempo. Dizemos que um bloco b é alcançável se houver um caminho de execução em $t_{\max}$ que comece em b e em que b seja completamente executado.

Para realizar essa busca, utilizamos um algoritmo baseado em uma variante do algoritmo de Dijkstra. Durante a procura, o tempo de execução de cada instrução é levado em conta conforme definido pelo modelo de custo adotado. A busca é iniciada a partir dos pontos de entrada de cada método e, em seguida, explora recursivamente os caminhos de execução possíveis, considerando os tempos de execução de cada instrução.

Durante a busca, o algoritmo acompanha o tempo acumulado de execução ao longo do caminho e verifica se esse tempo ultrapassa o limite $t_{\max}$. Se um bloco não puder ser completamente executado dentro do tempo limite, ele é excluído do conjunto de blocos alcançáveis. Ao final da busca, teremos identificado os blocos que são alcançáveis em $t_{\max}$, ou seja, aqueles que podem ser completamente executados dentro do limite de tempo. Esses blocos serão os alvos para o cálculo do WCEC de cada método.

6.3 Formulação do Problema ILP

Após ter sido definido o conjunto de blocos alcançáveis, é possível especificar as restrições que as execuções iniciadas por um bloco de entrada b_0 devem respeitar e, em seguida, encontrar a execução com o maior consumo que satisfaça essas condições. Para um conjunto de blocos $b_1, \cdots, b_n$, com tempos de execução $t_1, \cdots, t_n$ e consumos de energia $c_1, \cdots, c_n$, é possível expressar essas restrições na forma de um problema de Programação Linear Inteira (ILP) [2]. As variáveis $x_1, \cdots, x_n$ correspondem ao número de execuções de cada bloco. Deseja-se maximizar a soma $\sum c_i x_i$ enquanto se mantém a condição $\sum t_i x_i \leq t_{max}$.

Também é necessário garantir que o fluxo de controle do programa permaneça consistente: para uma execução, o número de arestas de entrada de um bloco deve ser igual ao número de arestas de saída do bloco, sendo esse número igual ao número de execuções do bloco. Denotaremos por $d_{i,j}$ a variável correspondente ao número de vezes que a aresta que conecta b_i a b_j é percorrida.

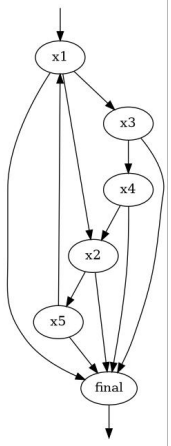
As arestas de entrada e saída devem ser percorridas exatamente uma vez. A aresta de entrada aponta para o bloco de entrada, e todos os blocos finais apontam para um bloco de retorno vazio, que possui apenas uma aresta de saída. Para um programa sem restrições de

tempo, os blocos finais são claramente definidos. Aqui, cada bloco pode ser o último bloco de uma execução em t_{max}, portanto, todos os blocos são finais.

A variável $d_{i,j}$ representa o número de vezes que a aresta de b_i para b_j é percorrida por uma execução.

A solução desse problema ILP será o número de execuções de cada bloco para o caminho que consome a maior energia dentro do conjunto de blocos alcançáveis, respeitando as restrições de tempo.

Essa análise simples ignora as condições de ramificação e os caminhos impossíveis em uma execução, e o resultado fornecido pela resolução do ILP pode ser uma sobreaproximação significativa do WCEC real. É necessário usar ferramentas de análise estática para refinar o ILP.



(a)

$$
\begin{aligned}
t_1x_1 + t_2x_2 + t_3x_3 + t_4x_4 + t_5x_5 &\leq t_{max} \\
x_1 = 1 + d_{5,1} &= d_{1,2} + d_{1,3} + d_{1,f} \\
x_2 = d_{1,2} + d_{4,2} &= d_{2,5} + d_{2,f} \\
x_3 = d_{1,3} &= d_{3,4} + d_{3,f} \\
x_4 = d_{3,4} &= d_{4,2} + d_{4,f} \\
x_5 = d_{2,5} &= d_{5,1} + d_{5,f} \\
x_f = d_{1,f} + d_{2,f} + d_{3,f} + d_{4,f} + d_{5,f} &= 1
\end{aligned}
$$

(b)

Figura 2: (a) Subgrafo dos blocos alcançáveis em t_{max}. (b) Construção de um problema ILP correspondente.

7 Obtenção de Mais Restrições para Maior Precisão

Aqui buscamos inferir estaticamente propriedades do programa para adicionar restrições ao ILP e refinar o WCEC final. Utilizaremos métodos de análise estática por interpretação abstrata [13] para obter limites sobre os valores dos resistores e o número de execuções de cada bloco em diferentes pontos do programa.

O analisador estático desenvolvido utiliza parcialmente essas técnicas e pode se beneficiar significativamente de uma utilização mais abrangente delas. No entanto, a análise é e permanecerá segura, visando possíveis melhorias futuras (consulte a seção 9).

Em vez de analisar todo o bytecode e tentar encontrar um ponto fixo, analisaremos separadamente cada um dos métodos, visando aplicabilidade em máquinas. A análise de cada método deve fornecer limites para o número de execuções de cada bloco após uma execução completa do respetivo método.

A escolha de analisar os métodos em vez de outros subprogramas é arbitrária, mas os

métodos são a partição mais simples do programa, seus pontos de entrada são explícitos e é fácil converter os limites encontrados em restrições para o problema ILP.

Supondo que tenhamos realizado a análise anterior em cada um dos métodos, teremos, para cada bloco b_i de um método m, um intervalo de limites $[l_i, u_i] \subseteq [0; +\infty]$, sendo o limite superior frequentemente infinito. Se um bloco b cuja variável associada no ILP é x invocar o método m, podemos adicionar a restrição $u_i \times x \leq x_i$ para modelar o fato de que cada execução de b levará a no máximo u_i execuções de b_i. Quanto aos limites inferiores, é preciso levar em conta que cada execução em t_{max} pode terminar em qualquer bloco de m, portanto temos a restrição $l_i \times (x - 1) \leq x_i$.

7.1 Análise Estática por Interpretação Abstrata

Para obter os limites mencionados, cada método é analisado estaticamente usando técnicas de interpretação abstrata. Introduções a essa teoria rica estão disponíveis em publicações como [1, 14]. Para um método m, o domínio D é definido como o produto dos domínios concretos dos registores e do número de execuções de cada bloco. Os valores dos registores são relevantes apenas para limitar o número de execuções. O domínio abstrato $D^\sharp$ é um domínio cuja especificação pode ser fornecida na implementação (geralmente usaremos um produto de intervalos, mas domínios mais precisos, como octógono, podem ser escolhidos). Uma semântica abstrata $[\![]\!]^\sharp : Op \to D \to D^\sharp$ também deve ser definida para todas as instruções do bytecode. Um ponto fixo é calculado a partir da semântica equacional do método, construída a partir do CFG.

Na prática, essa semântica geralmente se contenta em atribuir valores $\top$. De fato, não é possível conhecer o valor de retorno de um método externo chamado por meio de *invoke*, entre outros. Por enquanto, o analisador estático tem o propósito de obter limites para os métodos mais simples.

8 Implementação e *Benchmark*

O protótipo completo foi escrito em OCaml e está disponível no endereço `https://github.com/yarukha/wcec-dvk`. Ele inclui um *parser* completo do *bytecode* para uma representação interna a partir da qual o CFG é construído.

O ILP *solver* GLPK [15] é usado para definir e resolver os problemas ILP. Os CFGs são definidos usando Mapas (*Map*).

Para o analisador estático, mencionaremos o uso da biblioteca Apron [16], que permite representar domínios abstratos e desenvolver o analisador sem se preocupar com a implementação dos domínios. O calculador de pontos fixos Fix [17] é utilizado para obter um ponto fixo para cada método, construindo sua semântica equacional a partir do CFG.

Nossa avaliação experimental está em andamento com a participação do parceiro industrial envolvido no projeto em que este trabalho se encaixa. No momento, estamos em processo de integração deste analisador com um analisador alternativo de fundamentos mais tradicionais, desenvolvido pela mesma equipe. A expectativa é que essa integração resulte em uma análise mais abrangente. Os resultados experimentais obtidos através do uso do protótipo pelo parceiro industrial continuam em estágios iniciais, porém indicam a nossa capacidade de analisar aplicativos como Spotify, Facebook, entre outros. No entanto, os tempos de análise ainda não atendem às expectativas de um uso industrial em larga escala.

Para dar uma ideia deste ponto Foram realizados *benchmarks* em um aplicativo real simples contendo 9 MB de *bytecode*. Os modelos de custo utilizados (ver seção 4) não cobrem todo o *bytecode* utilizado nesse exemplo, especialmente métodos nativos não testados. Portanto, eles foram complementados de forma *razoável*, mas sem uma validação semelhante aos outros casos. Os resultados não têm valor para o consumo real do aplicativo, mas sim para seu consumo relativo.

t_{max}	WCEC	tempo de cálculo
5	5.3	28.4s
10	16.2	1m10s
20	35.6	2m25s
50	105	5m40
100	219	14m20s
150	324	25m36s
200	438	40m18s
500		>3h

Os modelos utilizados têm em média uma relação energia/tempo de 2, e uma instrução consome tipicamente uma unidade de energia. O número de instruções executáveis em t_{max} é aproximadamente $2t_{max}$. Os valores de WCEC têm um valor próximo de $2t_{max}$, coincidindo com a relação energia/tempo do modelo fornecido.

A análise dos N métodos gerará N problemas ILP, que são problemas NP-completos e cujo tamanho aumenta com t_{max}. O tempo de análise deve aumentar significativamente com t_{max}, e a análise de aplicativos pode se tornar impraticável para valores muito altos de t_{max}.

9 Conclusões e Trabalhos Futuros

A análise estática para estimativa do consumo de energia em aplicativos Android é um desafio técnico significativo, mas foi superado com sucesso através da definição de um modelo de custo e da implementação de um analisador estático parametrizado por esses modelos. Essa abordagem permitiu uma comparação precisa do consumo energético entre diferentes aplicativos Android.

O protótipo que incorpora essas ideias, apresentou resultados relevantes, destacando a eficácia da análise estática e do uso de modelos de custo na estimativa e comparação do consumo de energia dos aplicativos, indicando um caminho promissor para futuras pesquisas e desenvolvimentos nessa área.

Várias áreas de exploração surgem como possíveis trabalhos futuros:

- Definir modelos de tempo e energia mais precisos;
- Estudar detalhadamente os métodos nativos do Android e seu comportamento para abstrair seus valores de retorno. Técnicas de análise simbólica podem ser utilizadas [8];
- Desenvolver um analisador estático por interpretação abstrata para o Android, capaz de analisar grandes trechos de código com precisão.
- Formalizar o ambiente ART;
- Adaptar as técnicas atuais de WCET (análise de *cache* e pipeline, por exemplo) ao sistema Android.

Referências

[1] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, April 2008.

[2] Y.-T.S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, 1997.

[3] Enrico Eugenio and Agostino Cortesi. *WiFi-Related Energy Consumption Analysis of Mobile Devices in a Walkable Area by Abstract Interpretation*, volume 10109. January 2017. Pages: 39.

[4] Peter Wägemann, Tobias Distler, Timo Hönig, Heiko Janker, Rüdiger Kapitza, and Wolfgang Schröder-preikschat. Worst-Case Energy Consumption Analysis for Energy-Constrained Embedded Systems, 2015.

[5] Volkmar Sieh, Robert Burlacu, Timo Honig, Heiko Janker, Phillip Raffeck, Peter Wagemann, and Wolfgang Schroder-Preikschat. An End-to-End Toolchain: From Automated Cost Modeling to Static WCET and WCEC Analysis. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 158–167, Toronto, ON, Canada, May 2017. IEEE.

[6] James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Data Dependent Energy Modeling for Worst Case Energy Consumption Analysis. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*, pages 51–59, Sankt Goar Germany, June 2017. ACM.

[7] Jinseong Jeon, Kristopher K. Micinski, and Jeffrey S. Foster. Symdroid: Symbolic execution for dalvik bytecode. 2012.

[8] Erik Ramsgaard Wognsen, Henrik Søndberg Karlsen, Mads Chr. Olesen, and René Rydhof Hansen. Formalisation and analysis of dalvik bytecode. *Science of Computer Programming*, 92:25–55, 2014. Special issue on Bytecode 2012.

[9] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.

[10] Eric Bodden, Dr Andreas Zeller, and Dr Patrick Eugster. Vom Fachbereich Informatik der Technische Universität Darmstadt zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.) genehmigte Dissertation von Steven Arzt, M.Sc. M.Sc. aus Heidelberg. page 183.

[11] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis - SOAP '12*, pages 27–38, Beijing, China, 2012. ACM Press.

[12] Jianjun Zhao. Analyzing control flow in java bytecode. In *in Proc. 16th Conference of Japan Society for Software Science and Technology*, pages 313–316, 1999.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '77*, pages 238–252, Los Angeles, California, 1977. ACM Press.

[14] Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021.

[15] GLPK - GNU Project - Free Software Foundation (FSF). [Online] https://www.gnu.org/software/glpk/#TOCdocumentation. accessed: 2023-07-26.

[16] Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643, pages 661–667. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Series Title: Lecture Notes in Computer Science.

[17] François Pottier. Lazy least fixed points in ML. Unpublished, December 2009.