DOR :20.1001.1.2538161.1401.7.2.4.1

دریافت مقاله: ۱۴۰۱/۰۳/۱۰
پذیرش مقاله:۱۴۰۱/۰۵/۱۹

نوع مقاله: پژوهشی

بررسی تاثیر بازسازی‌های پرکاربرد کد برنامه‌های اندرویدی بر بهینه‌سازی مصرف انرژی

مهرداد آشتیانی*
استادیار دانشگاه علم و صنعت ایران، ایران، تهران
پست الکترونیکی: m_ashtiani@iust.ac.ir

علی بهروزی‌نیا
دانشجوی کارشناس ارشد دانشگاه علم و صنعت ایران، ایران، تهران
پست الکترونیکی: a_behrouzinia@comp.iust.ac.ir

نسرین حمزه لو
دانشجوی کارشناس ارشد دانشگاه آزاد اسلامی قزوین، ایران، قزوین
پست الکترونیکی: nasrinhamzelou@qiau.ac.ir

چکیده

با توجه به گسترش استفاده از دستگاه‌های همراه باتری‌محور از جمله گوشی‌های موبایل، لوحه‌ها، ساعت‌های نسل جدید و غیره، شرکت‌های بزرگ نرم‌افزاری که در زمینه تولید و توسعه برنامه‌های موبایلی فعالیت می‌کنند توجه ویژه‌ای به استفاده بهینه از باتری و منابع دستگاه پیدا کرده‌اند و این مقوله تبدیل به یکی از عوامل رقابتی در عرصه برنامه‌های موبایلی شده است. مقالات و پژوهش‌های زیادی در زمینه بازسازی کد نرم‌افزاری با هدف استفاده بهینه از باتری بخصوص در زمینه برنامه‌های اندرویدی انجام شده است که بعضاً منجر به تولید ابزارهای بازسازی نیز شده است. منظور از بازسازی کد، تغییر ساختار برنامه یا تغییرات در سطح کد برنامه به منظور تصحیح نابسامانی‌های کد، بدون تغییر رفتار و خروجی برنامه است. از جمله این ابزارها می‌توان به AutoRefactor، Leafactor و ابزارهای موجود در محیط‌های توسعه یکپارچه که به شکل افزونه هستند اشاره کرد. در این پژوهش، ۸ مورد از بازسازی‌های رایج موجود و پرکاربرد در زمینه توسعه برنامه‌های اندرویدی از لحاظ بهینه‌سازی استفاده از باتری و منابع مورد بررسی قرار گرفته‌اند. برخی از این بازسازی‌ها عبارتند از: Member، ViewHolder، Wake lock، Draw allocation ignoring method و Overdraw. در این پژوهش نشان داده شد که با انجام بازسازی بر روی نابسامانی‌های کد ذکر شده، مصرف انرژی می‌تواند بهبود پیدا کند. به‌عنوان مثال، بازسازی ViewHolder می‌تواند به بهبود ۱۳,۹ درصدی مصرف انرژی و بازسازی Overdraw می‌تواند به بهبود ۱۳,۲ درصدی منتهی شود. همچنین، با مقایسه با روش‌های کروز و دیگران و کوتو و دیگران نشان داده شد که الگوریتم‌های بازسازی پیشنهادی در این پژوهش بخصوص در مورد دو بازسازی ذکر شده توانسته‌اند

* نویسندهٔ مسئول

عملکرد بهتری در حوزه کاهش مصرف انرژی نشان دهند.
واژه‌های کلیدی: بازسازی، اندروید، نابسامانی کد، مصرف بهینه انرژی.

۱- مقدمه

امروزه با توجه به گسترش برنامه‌های اندرویدی و با توجه به این که برای هر کار بخصوصی برنامه‌های مشابه بسیاری یافت می‌شوند یکی از معیارهای انتخاب برنامه توسط کاربران، میزان انرژی مصرفی هر برنامه است [۱]. در حال حاضر جامعه تولیدکنندگان برنامه‌های اندرویدی به سمت بهینه کردن برنامه‌های تولید شده پیش می‌روند تا بدین ترتیب در رقابت با برنامه‌های مشابه، کاربران بیشتری را به سمت خود بکشند[۲].

اکنون، نابسامانی‌های کد بسیاری توسط تولیدکنندگان برنامه‌های اندرویدی شناسایی شده و روش‌های بسیاری برای بازسازی آن‌ها نیز توسط تولیدکنندگان باتجربه ارائه شده است. تولیدکنندگان بنا به تجربه از این روش‌ها استفاده می‌کنند تا بتوانند هر چه بیشتر برنامه‌های خود را بهبود بخشند[۳,٤]. سوالی که مطرح می‌شود این است که آیا بازسازی کد می‌تواند به بهبود مصرف انرژی برنامه‌های موبایلی کمک کند؟ ما در این مقاله با انتخاب ۸ مورد از نابسامانی‌های کد شناخته شده برنامه‌های موبایلی قصد داریم به این سوال جواب دهیم. هدف اصلی این مقاله این است که روش‌های بازسازی ۸ نابسامانی کد شناخته شده انتخابی را از نظر میزان بهبود مصرف انرژی برنامه‌های اندرویدی بررسی کرده و همچنین روش‌هایی برای رفع عیوب رویکردهای کنونی ارائه دهد. مقالات زیادی در زمینه بررسی تاثیر بازسازی نابسامانی‌های کد پرکاربرد بر میزان مصرف انرژی برنامه‌های اندرویدی نوشته شده است. هر یک از این مقالات سعی در ارائه یک روش برای بازسازی نابسامانی‌های کد دارد. ایرادهایی که به این روش‌ها وارد است در دو دسته جای می‌گیرند یکی ایرادات روش‌های بازسازی نابسامانی‌های کد و دیگری روش ارزیابی و سنجش میزان تاثیر هر یک از بازسازی‌ها.

در این مقاله، دو کار شناخته شده و پرارجاع کوتو و دیگران [۳] و کروز و دیگران [٤] در حوزه بازسازی کدهای نرم‌افزاری با رویکرد کاهش مصرف انرژی به دقت مورد بررسی قرار گرفته و نقاط ضعف هریک بیان شده است. در ادامه، یک روش جدید پیشنهاد می‌شود که سعی بر رفع ایرادهای موجود و بهبود عملکرد روش‌های پیشین دارد. در واقع، انگیزه اصلی این مقاله این است که بازسازی‌های مناسبی برای نابسامانی‌های کد پرکاربرد به تولیدکنندگان برنامه‌های اندرویدی ارائه شود و همچنین تاثیر این نابسامانی‌های کد به تولیدکنندگان نشان داده شود تا بتوانند معیاری برای انتخاب بازسازی مناسب جهت رسیدن به هدف مورد نظر داشته باشند.

در این مقاله، برای ۸ نابسامانی کد انتخابی در برنامه‌های اندرویدی که سبب مصرف بالای انرژی دستگاه می‌شوند روشی جدید برای بازسازی ارائه دادیم. همچنین، برای بررسی تاثیر روش‌های پیشنهادی برای بازسازی نابسامانی‌های کد مطرح شده روشی ارائه شده است تا از تاثیر عملکرد آن‌ها مطمئن شویم. در نهایت، نابسامانی‌های کد بررسی شده را بر اساس میزان تاثیر بازسازی آن‌ها رتبه بندی کردیم. به منظور ارزیابی کار پیشنهادی، از دو برنامه نمونه که حاوی نابسامانی‌های کد ذکر شده هستند استفاده شده است. مصرف انرژی این برنامه‌ها پیش و پس از بازسازی کد، مورد اندازه‌گیری قرار گرفته است. نشان داده شده است که بازسازی نابسامانی‌های کد Viewholder و Overdraw می‌توانند به ترتیب تا ۱۳,۹ و ۱۳,۲ درصد بر کاهش مصرف انرژی تاثیرگذار باشند. همچنین، بازسازیهای Wakelock و Draw Allocation نیز با ۷,٦ و ۳,۸ درصد بیشترین میزان اثرگذاری را داشته‌اند. متوسط میزان بهبود بازسازی کد به منظور تصحیح ۸ نابسامانی کد انتخاب شده معادل ٥,٤۲ درصد است. با مقایسه با روش‌های کروز و دیگران و کوتو و دیگران نشان داده شده است که الگوریتم‌های بازسازی

پیشنهادی بخصوص در مورد دو بازسازی Viewholder و Overdraw توانسته‌اند با عملکرد بهتری در حوزه کاهش مصرف انرژی نشان دهند.

در ادامه این مقالـــه، در بخش ۲ به مفاهیم پایه پرداخته شـــده، بخش ۳ کارهای مرتبط را بررســـی می‌کند. رویکرد پیشـــنهادی در بخش ٤ معرفی شـــده اســـت. ارزیابی کار پیشنهادی در بخش ٥ آورده شده است. در فصل ٦ جمع‌بندی و در نهایت در فصل ۷ کارهای آینده معرفی شده‌اند.

۲- مفاهیم پایه

در این بخش، مفاهیم پایه‌ای مرتبط را توضیح می‌دهیم تا برای ادامه مطالب این نوشته، به ادبیات مشترکی برسیم. ابتدا مفهوم نابســـامانی کد و بازســـازی را بیان می‌کنیم، سپس به بررسی نابســـامانی‌های کد انتخاب شده در این مقاله می‌پردازیم.

۲-۱ نابسامانی کد

ســـاختار خوب یک نرم‌افزار برای گسترش و نگهداری یک سیستم ضروری است. توســـعه نرم‌افزار یک فعالیت بی‌نظـــم و پراکنده اســـت. بنابرایـــن، نحوه پیاده‌ســـازی سیســـتم‌ها از ســـاختار برنامه‌ریـــزی شـــده و منطبق با معمـــاری، تجزیه و تحلیـــل و طراحی، منحرف می‌شـــود [٥]. تغییر ســـاختار نرم‌افزار، یک روش مؤثر برای بهبود آن اســـت. قابلیـــت انعطاف‌پذیری، نگهـــداری و دارا بودن ساختاری منســـجم، ویژگی‌های اصلی یک نرم‌افزار خوب هســـتند. برای این‌که یک نرم‌افزار دارای چنین ویژگی‌هایی بشود، نیازمند آن است که برنامه نویسان در همان مراحل اولیه نوشتن یک نرم‌افزار، طبق اصول و روش‌های خاصی مانـــند اصول شـــیءگرایی عمل کنند. متأســـفانه به خاطر کمبود وقت، سهل‌انگاری، کمبود نیرو و مشکلات مالی، در نوشـــتن بســـیاری از نرم‌افزارها به این اصول توجه کامل نمی‌شـــود. همین بی‌توجهی منجر به ایجـــاد یک نرم‌افزار غیر منســـجم با ساختاری اشتباه یا پیچیده یا گاه مشکل‌زا در آینـــده می‌شـــود. در علوم کامپیوتر، این ســـاختارهای نامناســـب به «نابسامانی کد» معروف هســـتند[٦،٨]. اغلب این نابســـامانی‌های کد در آینده تأثیر خود را روی برنامه می‌گذارنـــد و تقریباً تصحیح و تغییـــر یک برنامه را در آن زمان غیرممکن می‌کنند. بنابراین، توجه به اصول درســـت برنامه‌نویسی و شناسایی نابسامانی کد و تصحیح آن در همان قدم‌های اولیهٔ نوشته شدن یک نرم‌افزار، جزو وظایف اصلی یک برنامه‌نویس است. در ادامه، نابسامانی‌های کدی که در این مقاله آمده‌اند را بررسی می‌کنیم.

۲-۱-۱ استفاده از هش‌مپ

هش‌مپ بخشی از چارچوب زبان جاوا است. این مولفه، پیاده‌ســـازی رابط نقشـــه جاوا را فراهم می‌کند. داده‌ها را به‌صـــورت جفت (کلید، مقدار) ذخیره می‌کند که می‌توان با نمایـــه‌ای از نوع دیگری (به‌عنوان مثال یک عدد صحیح) به آن‌ها دسترسی یافت. یک شیء به‌عنوان یک کلید برای یک شیء دیگر اســـتفاده می‌شود. اگر بخواهیم کلیدی تکراری وارد کنیم، جایگزین کلید مربوطه می‌شود [۷].

۲-۱-۲ WakeLock

سازوکاری است که نشان می‌دهد برنامه باید دستگاه را روشـــن نگه دارد. هر برنامه‌ای که از آن استفاده می‌کند باید مجـــوز مربوط را در مانیفســـت برنامه درخواســـت کند. این ویژگی می‌تواند برای روشـــن نگه داشتن صفحه یا پردازنده هنگامی که دســـتگاه در حالت خواب اســـت به منظور انجام وظایف استفاده شود[۹]. اگر برنامه‌ای نتواند WakeLock را آزاد کند یا از آن بدون نیاز خاصی استفاده کند، می‌تواند باتری را تخلیه کند.

۲-۱-۳ عدم استفاده از Viewholder

مولفه Viewholder برای ایجاد یک پیمایش روانتر در نمای لیست استفاده می‌شـــود. هنگامی که در نمای لیست قرار دارید، سیستم باید هر مورد را ترسیم کند[۱۰]. برای کارآمدتر کردن این فرآیند، داده‌های ترســـیم شده قبلی را می‌توان مجددا مورد استفاده قرار داد. تعداد فراخوانی‌های تابع که به‌عنوان یک فرآیند هزینه‌بر شناخته می‌شود، با این روش کاهش می‌یابد.

۲-۱-۴ بازترسیم

یک برنامه ممکن اســت یک پیکســل را بیش از یک بار در یک قاب بکشد که به آن بازترسیم می‌گویند. بازترسیم معمولاً غیرضروری اســت و بهتر اســت حذف شود. این رخداد خود را به‌عنوان یک مشــکل عملکرد با اتلاف زمان پردازنده گرافیکی برای ترســیم پیکسل‌هایی که به چیزی که کاربر روی صفحه می‌بیند کمکی نمی‌کند نشان می‌دهد. بازترســیم به ترسیم چندین پیکســل روی صفحه توسط سیستم در یک قاب اشــاره دارد. همان‌طور که در شکل ۱ می‌بینیم رنگ‌هایی که به رنــگ قرمز نزدیک‌تراند به معنی قســمت‌هایی از صفحه‌اند که دچار بازترســیم شده‌اند. از ســوی دیگر، رنگ‌هــای مایل به آبی در صفحه نشــان از قسمت‌هایی‌اند که بازترسیم در آن‌ها رخ نداده است.

۲-۱-۵ استفاده از عناصر منسوخ

منابعی مانند نمادها یا عناصر رابط کاربری ممکن است به دلیل تغییرات در نرم‌افزار منســوخ شــوند. با این حال، تولیدکنندگان ممکن اســت فراموش کنند که آن‌ها را از کد منبــع برنامه حذف کنند که این موضوع اندازه برنامه‌ها را بزرگ‌تر می‌کند و در نتیجه سرعت کامپایل و ساخت آن‌ها را کاهش می‌دهد[۷، ۱۱].

۲-۱-۶ تخصیص هنگام ترسیم

این یک روش نامناســب برای ســاخت اشیاء در طول عملیات طراحی یا چیدمان است. تخصیص اشیاء می‌تواند باعث عملیات جمع‌آوری اشــیاء بدون اســتفاده شود که ســرعت عملیات را کاهش می‌دهد و یک رابط کاربری غیر روان ایجاد می‌کند. راه‌حل پیشــنهادی، تخصیص اشیاء از قبل و اســتفاده مجدد از آن‌ها برای هر عملیات ترســیمی است[۱۲].

۲-۱-۷ MIM

منظور از این نابســامانی کد وجــود یک تابع غیرثابت در داخل یک رده اســت، که در عوض می‌تواند ثابت باشد. یعنی به هیچ متغیری از رده دسترســی ندارد و مســتقیماً روش‌های غیرثابت را فراخوانی نمی‌کند. توابع ثابت در یک

شکل ۱: نمای بازترسیم در یک برنامه اندرویدی[۵].

بخش حافظه جدا از محل ذخیره اشــیاء ذخیره می‌شــوند و مهم نیســت که در طول اجرای برنامــه چند نمونه رده ایجاد شــده اســت. فقط یک نمونه از چنین تابعی ایجاد و استفاده می‌شود. این ســازوکار به کاهش مصرف انرژی کمک می‌کند.

۲-۱-۸ عدم استفاده از تابع recycle

با توجه به این که در برنامه‌نویسی اندروید در مواردی نیاز است متغیرهای مربوط به فایل منبع را در کد بخوانیم و بنابر مقدار آن‌ها در ادامه تصمیم‌گیری‌های لازم را انجام دهیم، برای خواندن این مقادیر باید از TypedArray استفاده کنیــم. با توجه به این که با اســتفاده از این نوع می‌توانیم به منابع دسترســی پیدا کنیم و از آنجایی که نگه داشــتن یــک نمونه از آن فضای زیادی در حافظه اشــغال می‌کند، می‌تواند در بحث مدیریت حافظه مشکل آفرین شود و باید با استفاده از تابع recycle فضای اشغال شده را آزاد نمود.

۲-۲ بازسازی

بازســازی کد فراینــد تغییر یک سیســتم نرم‌افزاری به‌گونه‌ای است که رفتار خارجی آن تغییر نکند و ساختار داخلی بهبود یابد. این یک روش منظم برای پاک‌ســازی کد به‌گونه‌ای اســت که شــانس رخ دادن خطا کاهش یابد. در واقع، بازسازی یا بازسازی کردن معادل بهبود طراحی کد بعد از نوشــته شدن است. با توجه به این‌که نرم‌افزار دائم در حال تغییر و گســترش است، همیشــه احتمال دارد در مرحله‌ای دچار نابســامانی کد شود. روش‌های بازسازی نابسامانی کد کمک می‌کنند نابسامانی‌های کد ایجادشده را بدون تغییرات درونی از بین ببریم[۱۲،۱۳].

با توجه به این‌که هر نابسامانی کد یک ساختار مشخص دارد، بـرای هـر کدام، یک روش بازسـازی کـد در نظر گرفته‌شده است. برنامه‌نویس اگر بخواهد چه دستی چه با ابزار، سـاختارهای نامناسب کد را پیدا و سپس بازسازی کند می‌تواند دچار اشـتباه در شناسـایی نابسامانی کد و خطا در تصحیح برنامه بشود. چرا که هیچ ابزاری تضمین کامل برای شناسـایی و بازسازی درست را نمی‌دهد[١٤]. به‌عبارتی‌دیگر هر ابزاری مناسـب شناسایی و بازسازی نابسامانی‌های کد خاصی اسـت و دقت هر ابزار و روش در نابسامانی‌های کد مختلف فرق دارد.

۲-۳ نمایه‌گر انرژی

نمایه‌گـر انرژی کمک می‌کند تا جاهایی که برنامه بیش از حـد لازم انرژی مصرف می‌کند، پیدا شـوند. نمایه‌گر، اسـتفاده از پردازنده، شبکه وسایر منابع را نظارت می‌کند و تصویـری از میزان مصرف انـرژی هر یک از این اجزا را نمایـش می‌دهد. نمایه انرژی، همچنیـن رویدادهایی از سیسـتم را نشـان می‌دهد که می‌توانند بر مصرف انرژی تأثیـر بگذارنـد. نمایه‌گر انرژی به‌طور مسـتقیم مصرف انرژی را اندازه‌گیری نمی‌کند. بلکه از مدلی استفاده می‌کند که مصرف انرژی را برای هر منبع روی دسـتگاه تخمین می‌زند. نمایه‌گر انرژی کمک می‌کند تا بتوان در مورد نحوه استفاده از هر کدام الگوها تصمیمات آگاهانه گرفت.

۲-۳-۱ Battery Historian

ابزاری اسـت که توسـط گوگل در سال ۲۰۱٦ معرفی شـد. این ابزار قابلیت تحلیل رفتار دسـتگاه تلفن همراه و به‌طـور دقیق‌تر تحلیـل اطلاعات و رخدادهـای مرتبط به باتری دستگاه را فراهم می‌آورد. یکی از استفاده‌های مهم این ابزار در یافتن الگوهای پرمصرف انرژی در تلفن همراه است. همچنین، بینشی در مورد مصرف باتری دستگاه در طول زمان ارائه می‌دهد. در سـطح کل سیسـتم، این ابزار رویدادهـای مرتبـط با انـرژی را از پرونـدهٔ رخدادهای سیسـتم در یک نمایش تحت وب تصویر می‌کند. در سطح ویـژه برنامه، این ابزار، داده‌هـای مختلفی را ارائه می‌کند که می‌تواند در شناسایی رفتار برنامه‌ای که انرژی زیادی مصـرف می‌کند کمک کند. این ابزار، تجسـمی در سـطح سیسـتم از رفتارهای مختلـف برنامه و سیسـتم همراه با همبسـتگی آن‌ها با مصرف باتـری در طول زمان ارائه می‌دهد. این نما می‌تواند در تشخیص و شناسایی مشکلات مصرف انـرژی در برنامه کمک کند. عـلاوه بر داده‌های سطح کلان ارائه شده توسـط نمای گسترده سیستم، این ابزار همچنین جداول و برخی تجسـم داده‌های خاص هر برنامـه‌ای را که در دسـتگاه اجرا می‌شـود ارائه می‌دهد. داده‌های جدولی شامل موارد زیر هستند:

١- میزان مصرف انرژی تخمینی برنامه در دستگاه.
٢- اطلاعات شبکه.
٣- Wakelockها.
٤- خدمات.
٥- اطلاعات فرآیند.

۳- کارهای مرتبط

در بخش قبلی به تعریف و بررسـی مفاهیم پایه مورد نیـاز پرداختیم. در این بخش سـعی می‌کنیـم مروری بر کارهای مرتبط در حوزه بازسـازی نابسامانی‌های کد در برنامه‌های اندرویدی و همچنین بررسـی میزان تاثیر این بازسازی‌ها بر میزان انرژی مصرفی برنامه‌های اندرویدی داشـته باشـیم. در این بخش، رویکرد و فرایندها و نتایج کارهـای مشـابه را معرفی و بررسـی می‌کنیم. در همین راسـتا، دو روش شناخته شـده مختلف را در زمینه تاثیر بازسازی نابسامانی‌های کد بر انرژی مصرفی برنامه‌های اندرویدی بررسی می‌کنیم.

۳-۱ روش کوتو و دیگران

در این قسـمت بـه بررسـی روش کوتـو و دیگران می‌پردازیـم [۳]. در این مقاله ابتدا چند نابسـامانی کد که توسط مقاله‌های دیگری بررسی شده و بهینه نبودن آن‌ها نشـان داده شده است معرفی می‌شـوند. کوتو و دیگران

نـــام EGAP [1] را بر این نابســـامانی‌های کـــد می‌گذارند. در ادامه، آن‌ها برای هـــر یک از این EGAPها یک راه‌حل برای بازســـازی پیشـــنهاد می‌کنند که در ادامه به بررسی نتایج تاثیر این بازســـازی‌ها می‌پردازیم. موارد بررسی شده در این مقاله عبارت‌اند از:

- Draw Allocation
- Wake Lock
- عدم استفاده از recycle()
- Obsolete layout params
- عدم استفاده از ViewHolder
- استفاده از HashMap
- Excessive Method Calls
- Member Ignoring Method
- Overdraw

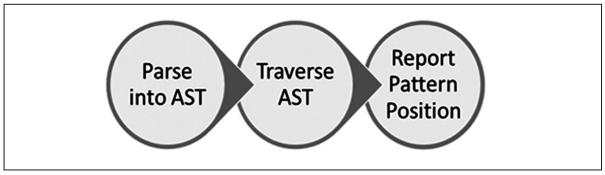
برای بررسی تاثیر بازسازی‌ها بر روی نابسامانی‌های کد از تعداد زیادی برنامه اندروید اســـتفاده شده است که ایـــن برنامه‌ها از طریق MUSE به‌دســـت آمده‌اند. مجموعه داده MUSE یک مجموعه از برنامه‌های اندرویدی به زبان جاوا است. این پروژه‌ها از بُن‌سازه‌های معتبری مانند گیت هاب[2] و گیت لب[3] اســـتفاده می‌کند. از ســـوی دیگر، MUSE یـــک پایگاه داده از اطلاعات اضافی راجع به پروژه‌ها دارد. اطلاعاتی مانند تعداد فایل‌ها و تعداد رده‌ها. در گام بعد ۶۰۹ پروژه انتخاب می‌شوند و پارامترهای اولیه آن‌ها محاسبه می‌شوند. این اطلاعات را در جدول ۱ مشاهده می‌کنیم.

در گام بعدی کوتو و دیگران روشـــی پیشنهاد می‌کنند که به‌وســـیلهٔ آن بتوانند نابسامانی‌های کد را در برنامه‌ها تشـــخیص دهند. الگوریتم تشخیص در شکل ۲ نشان داده شده است. طبق این الگوریتم، برنامه اندرویدی ابتدا تبدیل به درخت نحوی انتزاعی می‌شود. در ادامه روی این درخت نحوی انتزاعی پیمایش انجام شـــده و هر جا که الگوی یک نابســـامانی کد مورد نظر یافت شد آن را مشخص خواهد کرد. این فرآیند، توســـط ابزاری به نـــام لینت انجام گرفته اســـت. این ابزار این امکان را فراهـــم می‌کند که با تعریف

1- Energy Greedy Android Pattern
2- GitHub
3- GitLab

جدول ۱: نمایش معیارهای برنامه‌ها [۳]

	Java Files	XML Files	Classes	Methods	LOC
Min	2	1	2	1	22
Max	3492	4929	4267	39511	668085
Average	73	407	65	524	10822
TOTAL	44959	248385	39926	319636	6590636



شکل ۲: الگوریتم تشخیص نابسمانی‌های کد.

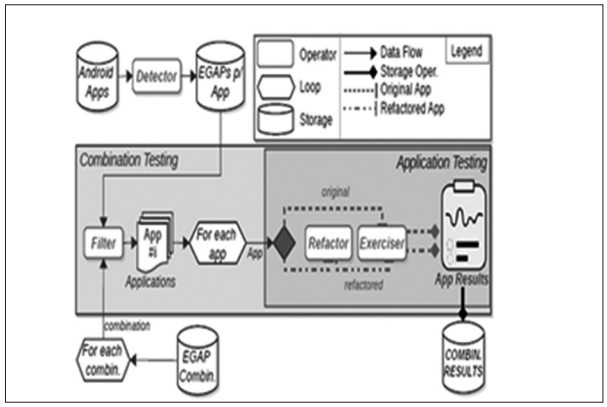
```
private void checkAndReport(PsiVariable var) {
  if (var.getType() == null) return;
  String varType = var.getType().getCanonicalText();
  PsiExpression init = var.getInitializer();

    if (varType.startsWith(mHashMapClass)) {
      Reporter.reportIssue(mContext, ISSUE, var);
    } else if (varType.startsWith(mMapClass)
      && init != null && init.getType() != null) {
        String initTp = init.getType().getCanonicalText();
        if (initTp.startsWith(mHashMapClass)) {
          Reporter.reportIssue(mContext, ISSUE, var);
        }
} }
```

شکل ۳: قاعده نوشته شده برای یافتن الگوی هش‌مپ[۳].

مجموعه‌ای از قواعد[4] بتوانیم الگوی‌های شخصی‌ســـازی شـــده را بیابیـــم. بدیـــن ترتیـــب، در گام بعدی بـــرای هر نابســـامانی کد یک قاعده تعریف و نوشـــته شده است. در ادامه، یک نمونه قاعده نوشـــته شده برای یافتن نابسامانی کد HashMap را در شکل ۳ مشاهده می‌کنیم.

در گام بعدی با در دست داشتن این قواعد و ابزار لینت، ۶۰۹ برنامه موجود مورد بررســـی قرار گرفته و اطلاعاتی ماننـــد تعـــداد EGAP های هر برنامه، انـــواع آن و محل آن در یک فایل جیســـون[5] نوشـــته و ذخیره شده است. آن‌ها برای تصدیق صحت عملکرد قواعـــد یک آزمایش طراحی کرده که بدین شـــکل اســـت که به چند برنامـــه اندرویدی به‌صورت دستی الگوی نابسامانی‌های کد را وارد کرده و سپس با اســـتفاده از لینت آن‌ها را جستجو می‌کنند. نتیجه این آزمایش این اســـت که تمام الگوهای وارد شـــده بدون خطا پیدا شـــده‌اند. در قدم بعدی آن‌ها با اســـتفاده از ابزار

4- Rule
5- Json



شکل ۴: فرآیند انجام آزمایش در روش کوتو و دیگران[۳].

AutoRefactor کــه به‌صورت یک افزونه[6] در Eclipse IDE قرار دارد نابسامانی‌های کد یافت شده را بازسازی می‌کنند. از ۶۰۹ برنامه مورد بررســی ۲۳۹ تــای آن‌ها حداقل یک EGAP داشتند. در ادامه، با روش و فرایندی که این پژوهش برای سنجش تاثیر بازســازی‌ها بر میزان مصرف انرژی برنامه‌های اندرویدی طی کرده آشنا می‌شویم. همان‌طور که از شــکل ٤ مشخص اســت چهار عملیات اصلی وجود دارد که با مستطیل‌های سبز رنگ مشخص شده‌اند.

۱- Detector: وظیفــه تشــخیص EGAP هــا را دارد. ورودی آن برنامه‌هــای اندرویدی و خروجی آن اطلاعات EGAP های یافت شــده است که در یک پایگاه داده ذخیره می‌شود.

۲- Filter: وظیفه پالایش‌کــردن برنامه‌های اندرویدی با ترکیب‌های مختلف EGAP را دارد. دو نســخه از ترکیب EGAP برای هر برنامه ایجاد می‌شــود که یکی نســخه با EGAP و دیگری نسخه اصلی بدون تغییر است.

۳- Refactor: وظیفه بازســازی EGAP های مورد نظر را دارد.

٤- Exerciser: وظیفه اجــرای برنامه‌های اندرویدی را دارد تا با اجرای ســناریوهای مشخص شده میزان انرژی مصرفی آن را اندازه بگیرد.

همان‌طــور کــه در نمودار ســمت چپ شــکل ۵ دیده می‌شود. خطوط قرمز تعداد آزمایش‌هایی را نشان می‌دهد که در آن بازسازی منجر به افزایش میزان مصرف انرژی شده و خطوط ســبز تعداد آزمایش‌هایی را نشان می‌دهد که در آن بازســازی منجر به کاهش مصرف انرژی شده است. در نمودار سمت راستی درصد میزان تغییر مصرف انــرژی را می‌بینیم. با توجــه به نمودار شــکل ۵ متوجه می‌شویم که بازسازی نابســامانی‌های کد به‌صورت تکی همیشه تاثیر مثبتی بر میزان مصرف انرژی می‌گذارند. از ســوی دیگر اکثر ترکیب‌های بازسازی نابسامانی‌های کد نیز باز هم تاثیر مثبتی بر میزان مصرف انرژی می‌گذارند اما این تاثیر به‌طور میانگین نسبت به تاثیر بازسازی فردی نابســامانی‌های کد کمتر اســت. اما در تعداد اندکی از این ترکیب‌ها میزان بازســازی تاثیر منفــی بر میزان مصرف انرژی گذاشته است.

با استناد به این نتایج می‌توان اینطور نتیجه‌گیری کرد که بازســازی فردی نابســامانی‌های کد تاثیر مثبت معنی داری بر میزان مصرف انرژی دارد اما ترکیب بازســازی چند نابســامانی لزومــاً تاثیر مطلوبی بــر میزان مصرف انرژی نخواهد داشت.

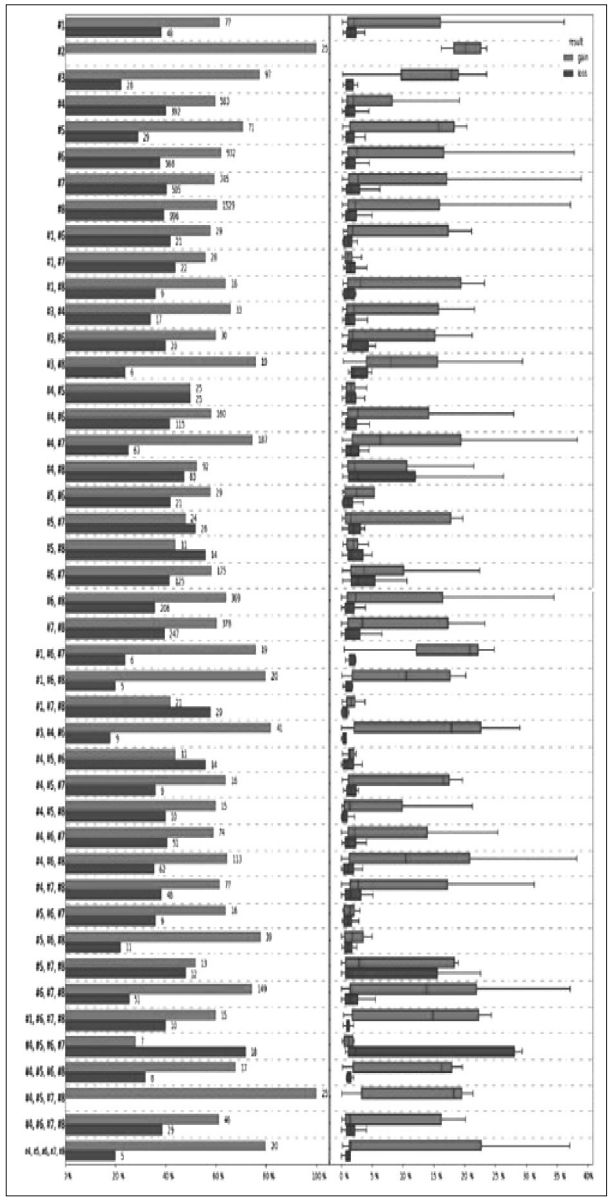
۳-۲ روش کروز و دیگران

در این قســمت، به معرفی روش کــروز و دیگران در بررسی تاثیر بازسازی برخی نابسامانی‌های کد بر میزان مصرف انرژی برنامه‌های اندرویدی می‌پردازیم] ٤[. کروز و دیگران اهداف خود از مقاله ارائه شده را این چنین اعلام می‌کننــد که ۱- فراهم آوردن مجموعــه‌ای از اصول برای توســعه برنامه‌هایی که انــرژی را بهینه مصرف می‌کنند. ۲- بررســی تاثیر هر یک از این اصول بر میزان مصرف انرژی در برنامه‌های اندرویدی.

در ادامــه، کروز و دیگران معیار مصرف انرژی را یک متغیر مســتقل و همچنین بهینه‌سازی برنامه را یک معیار وابســته قلمداد می‌کنند. نابســامانی‌های کدی که کروز و دیگران بررسی کرده‌اند عبارت‌اند از:

- Draw Allocation
- Wake Lock
- عدم استفاده از () recycle
- Obsolete layout params

6- Extension

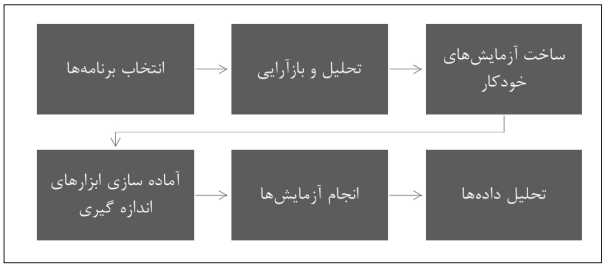


شکل ۵: نمای کلی نتایج آزمایشات کوتو و دیگران[۳].

- عدم استفاده از ViewHolder
- Overdraw
- Unused Resources
- Useless Parent
- Member Ignoring Method
- استفاده از HashMap

این مقاله مطابق فرآیندی که در شـــکل ٦ می‌بینیم پیش می‌رود. در ادامه هر یک از این مراحل را بررسی می‌کنیم.

در قسمت انتخاب برنامه‌ها، کروز و دیگران برای این‌که به منبع برنامه‌ها دسترسی داشته باشند از f-droid استفاده



شکل ۶:- فرآیند پیشبرد در روش کروز و دیگران[۳].

کرده‌اند و از محبوبیت برنامه‌ها در مارکت‌های برنامه‌های اندرویدی به‌عنوان معیار اولیه انتخاب برنامه‌ها اســـتفاده کرده‌انـــد. معیارهای دیگـــر برای انتخـــاب برنامه‌ها، منبع باز بودن و نداشـــتن عملیات ســـنگین در برنامه است. در نهایت ٦ برنامه را انتخـــاب کرده‌اند. اطلاعات و معیارهای برنامه‌های انتخاب شده را می‌توان در جدول ۲ دید. ستون اول، نام برنامه‌ها اســـت. ستون دوم تعداد نصب برنامه‌ها در مارکت‌ها اســـت. ستون سوم نمره برنامه در مارکت‌ها اســـت. ســـتون چهارم تعداد نفراتی را نشان می‌دهد که به برنامه نمره داده‌اند. ستون پنجم تعداد خط‌های هر برنامه است. ستون ششم تعداد فایل‌های منبع هر برنامه را نشان می‌دهد. ستون هفتم تعداد رده‌های هر برنامه است و ستون هشـــتم نیز میزان پیچیدگی برنامه را نشان می‌دهد. برای تشخیص و همچنین بازسازی نابسامانی‌های کد از ابزاری به نام لینت اســـتفاده می‌شـــود. این ابزار در کیت توسعه اندروید وجود دارد و می‌توان از آن استفاده کرد. از طرفی، نابسامانی‌های کد انتخاب شـــده بر اساس معیارهای زیر انتخاب شده‌اند:

۱- در برنامه‌های اندروید متداول باشند.

۲- قسمت‌های مهمی از برنامه را شامل شوند.

در ادامـــه با اســـتفاده از لینت این نابســـامانی‌های کد تشخیص داده شدند و برای هر نابسامانی کد برنامه مورد نظر به‌طور دســـتی بازسازی شد. همین‌طور یک نسخه از هر برنامه که همه نابسامانی‌های کد در آن بازسازی شدند هم ایجاد شد.

جدول ۲: معیارهای برنامه‌های استفاده شده در روش کروز[۴].

Application	Installs	Ratings	#ratings	LOC	LOC(XML)	Classes	CC
Habit Tracker	50000-100000	4.7	1252	28295	7302	193	2471
Writeley Pro	1000-5000	4.3	84	3251	2612	86	498
Talaramo	1000-5000	4.4	63	1043	192	26	131
Gnu Cash	50000-100000	4.3	2460	26532	20757	286	2846
Acrylic Paint	n.a	n.a	n.a	961	384	18	119
Simple Gallery	1000-5000	4.7	18	2227	685	37	434

Algorithm 1 *Loop - Habit Tracker* interaction script

```
1: SkipIntroductoryTips()
2: for i ← 1 to 10 do
3:     for i ← 1 to 7 do
4:         CreateNewHabit(i)
5:         CheckHabitDetails(i)
6:         ScrollThroughTheReport()
7:         GoBack()
8:     end for
9:     DeleteAllHabits()
10: end for
```

شکل ۷: نمونه‌ای از کد سناریو [۴].

از آنجایی که انرژی مصرفی در برنامه‌های اندروید به عوامل مختلفی بستگی دارد، برای اطمینان از درستی نتایج آزمایش آن‌ها را چند بار تکرار می‌کنند. برای اندازه‌گیری میزان انرژی، برای هر برنامه یک فایل آزمون[۷] نوشته می‌شود تا بدین ترتیب یک سناریو را بارها در برنامه اجرا کند. برای نوشتن این فایل از یک کتابخانه پایتون[۸] استفاده شده است که از طریق شناسه نمایه‌ها[۹] به مولفه‌های اندرویدی دسترسی می‌یابد و با آن‌ها ارتباط برقرار می‌کند. در شکل ۷ می‌توان یک نمونه از این فایل‌ها را دید.

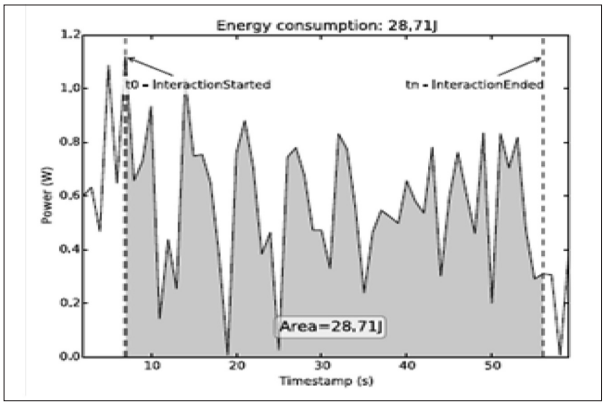
برای اندازه‌گیری میزان مصرف انرژی برنامه‌های اندرویدی از یک کامپیوتر bare-board استفاده می‌شود که دارای مشخصات زیر است:

– مدل: ODROID-XU

– نسخه اندروید: ٤.٢.٢

این دستگاه بسیار شبیه به یک دستگاه تلفن همراه است با این تفاوت که برخی مولفه‌ها را می‌توان در آن به‌طور کامل غیرفعال کرد تا بدین ترتیب اثر آن‌ها را به‌طور قطعی

7- UI Test
8- Android View Client
9- View ID



شکل ۸: میزان انرژی مصرفی یک برنامه[۴].

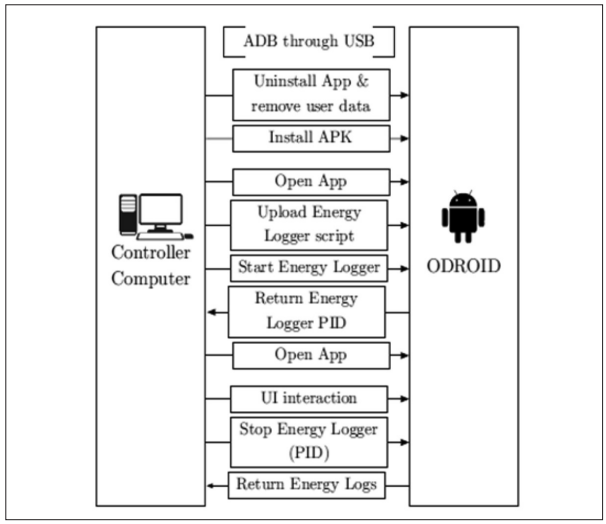
از نتایج آزمایش‌ها حذف کرد. در این قسمت، به نحوه محاسبه معیارهای مورد نظر کروز و دیگران می‌پردازیم. انرژی کل اندازه‌گیری شده بین دو زمان t_0 و t_n از طریق فرمول زیر محاسبه می‌شود به‌طوری که W انرژی مصرفی و P توان است.

$$w = \int_{t_0}^{t_n} p(t)\,dt \qquad (۱)$$

نموداری از فرمول بالا را در شکل ۸ مشاهده می‌کنیم که از طریق جمع فضای زیر نمودار به میزان انرژی مصرفی می‌رسیم.

$$\int_{t_0}^{t_n} p(t)dt \approx \frac{\Delta t}{2}[p(t_0) + 2p(t_1) + \cdots 2p(t_{n-1}) + p(t_n)] \qquad (۲)$$

شکل ۹ روند پیش‌برد هر یک از آزمایش‌ها را نشان می‌دهد. همان‌طور که از شکل برمی‌آید، هر آزمایش، مستقل از آزمایش دیگر است. ابتدا دستگاه به وسیله کابل USB به سیستم کنترل متصل می‌شود. سپس برنامه قبلی در صورت وجود روی دستگاه حذف و برنامه مورد نظر بر روی آن نصب می‌شود. سپس فرآیند پروفایل



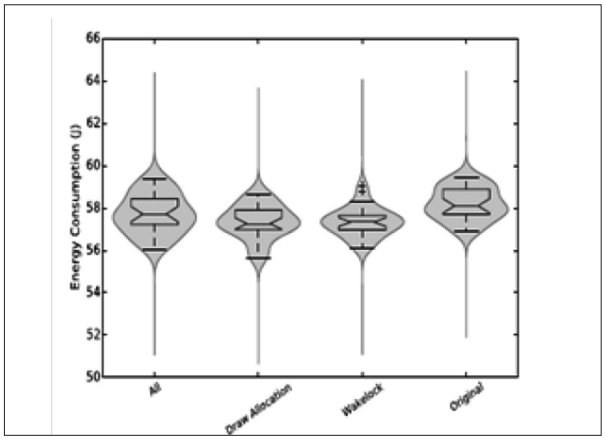
شکل ۹: فرآیند پیش‌برد آزمایش در روش کروز [۴].

انـــرژی صورت می‌گیرد. به این ترتیب که ابزار پروفایل با بازشدن برنامه شـــروع به کار می‌کند. در مرحله اول، یک برنامه ســـاده و خالی نصب می‌شود و بدین ترتیب میزان تقریبی انرژی مصرفی توسط خود سیستم عامل اندروید را می‌توان محاســـبه کرد. در مرحلـــه بعدی برنامه اصلی نصب می‌شود و هر سناریوی نوشته شده برای آن ۳۰ بار اجرا می‌گردد. در نهایت، همه خروجی‌ها که شـــامل میزان انرژی مصرفی و نســـخه برنامه است ذخیره می‌شود. طی بررسی داده‌های ذخیره شده در گام قبلی، کروز و دیگران متوجه داده‌هایی می‌شوند که نسبت به بقیه داده‌ها یا خیلی کوچک هســـتند و یا خیلی بـــزرگ. آن‌ها تحلیل می‌کنند که ایـــن داده‌های پرت به خطاهای سیســـتمی و یا رخدادهای غیرقابل پیش‌بینی مربوط هستند. خطاهایی که باعث بسته شدن برنامه می‌شـــوند و یا دیالوگ‌هایی که هنگام اجرای آزمایش توسط سیستم عامل باز می‌شوند.

کروز و دیگران برای کاهـــش تاثیر این داده‌های پرت، داده‌های خارج از بازه زیر را حذف می‌کنند.

$$[\bar{x} - 2s, \bar{x} + 2 \quad (۳)$$

در فرمول بالا $\bar{x}$ میانگین و S هم انحراف معیار اســـت. کـــروز و دیگـــران نتایج آزمایش‌هـــا را در قالب نمودار به تصویر می‌کشند که یک نمونه آن را در شکل ۱۰ می‌بینیم. در این نمودار می‌توان توزیع نرمال را مشاهده کرد. نتایج



شکل ۱۰: انرژی مصرفی نابسامانی‌های کد برای یک برنامه [۴].

آزمایش نشـــان داده اســـت که بازســـازی نابسامانی‌های کـــد WakeLock، OLP، DrawAllocation، ViewHolder و Recycle بیشـــترین تاثیـــر بـــر میزان مصـــرف انرژی برنامه‌های اندرویدی را دارند. جدول ۳ تاثیر چشم‌گیر هر نابســـامانی‌های کد را بر میزان مصـــرف انرژی برنامه‌ها نشـــان می‌دهد. در جدول زیر MD به معنی تفاوت میانگین است که از فرمول زیر به‌دست می‌آید.

$$MD = [\bar{x}_f - \bar{x}_o] \quad (٤)$$

در ایـــن فرمول x_f برابـــر با میانگین انـــرژی مصرفی نسخه بازسازی شده و x_o برابر با میانگین انرژی مصرفی برنامه اصلی اســـت. همچنین در جدول ۳ از معیار Cohen هم اســـتفاده شده اســـت. این روش برای تعیین اندازه اثر تفاوت اســـتاندارد دو میانگین به کار می‌رود که از فرمول زیر به‌دست می‌آید.

$$d = \frac{\bar{x}_f - \bar{x}_o}{s} \quad (٥)$$

در ایـــن فرمول s برابر اســـت با انحـــراف معیار. طبق تعریف، زمانی اندازه اثر بزرگ اســـت که مقدار d به‌دست آمده از ۰.۵ بیشتر باشـــد. بنابراین، در جدول هم مقدار d همه موارد از ۰.۵ بیشتر است چرا که همان‌طور که ذکر شد مواردی در جدول آورده شده که اثربخشی آن‌ها بالا است. معیار بعدی در جدول زیر IMP است که میزان اثربخشی را از رابطه زیر محاسبه می‌کنیم.

$$d = \frac{\bar{x}_f - \bar{x}_o}{\bar{x}_o} d = \frac{\bar{x}_f - \bar{x}_o}{\bar{x}_o} \quad (٦)$$

همچنین ستون آخر میزان دقایق صرفه جویی شده بر

جدول ۳: میزان تاثیر بازسازی‌های تاثیرگذار[۴].

Application	Pattern	MD	Cohen's d	(%) IMP	Savings ((rain
Writeily Pro	View Holder	-5.39	-0.78	4.50	65
	All	-5.42	-0.76	4.53	65
Talalarmo	DrawAllocation	-0.86	-1.11	1.47	21
	WakeLock	-0.85	-1.17	1.46	21
	All	-0.48	-0.57	0.82	12
GnuCash	ObsokteLayoutParam	-1.41	-0.67	0.72	10
	Recycle	-1.28	-0.66	0.65	9
	All	-1.53	-0.64	0.78	111
Acrylic Paint	Overdraw	1.42	1.64	-2.26	-33
	All	1.37	1.51	-2.18	-31
Simple Gallery	Overdraw	3.08	1.04	-2.11	-30

جدول ۴: مقایسه کلی روش‌های مرتبط

نام روش	کوتو و دیگران	کروز و دیگران
استفاده از آزمایش خودکار	خیر	بله
ترکیب‌های مختلف نابسامانی‌های کد	بله	خیر
بازسازی خودکار	بله	بله
جامع بودن برنامه‌ها انتخابی	بله	بله

اثر بازسازی یک نابسامانی کد را بعد از اجرای سناریوی آزمایشی برای ۲٤ ساعت کامل نشان می‌دهد.

در این بخش سعی کردیم که دو روش اصلی و پرارجاع برای بررسی نابسامانی‌های کد به منظور کاهش مصرف انرژی را بررسی کنیم. در واقع سعی شد که ایده‌های هر دو روش کروز و دیگران و کوتو و دیگران شرح داده شوند و قسمت‌های مهم آن‌ها با جزئیات بررسی شوند. همین‌طور نتایج بررسی‌ها و آزمایش‌های آنان و همین‌طور روش‌هایی که برای ارزیابی روش‌های خود استفاده کرده بودند مورد بررسی قرار گرفتند. هر دو روش مزایا و معایب خود را داشتند که در ادامه خلاصه آن‌ها را در جدول ٤ می‌بینیم.

لانونه و همکاران با تمرکز بر تعدادی از نابسامانی‌های کد تاثیرگذار بر مصرف انرژی برنامه‌های موبایلی، افزونه‌ای را برای اندروید استودیو طراحی کرده‌اند که این امکان را به تولیدکنندگان می‌دهد که بتوانند به‌صورت خودکار کد برنامه خود را از منظر وجود این نابسامانی‌ها بررسی کنند [١٦]. اگرچه اینکار به دلیل معرفی یک افزونه با کارکرد خودکار بسیار ارزشمند است اما تعداد نابسامانی‌های انتخاب شده محدود بوده (٤ نابسامانی) و همچنین توضیحات بسیار کلی و مختصری از چگونگی اعمال الگوریتم‌های بازسازی داده شده است.

کروز و آبریو در مقاله خود ابزاری به نام Leafac-tor را معرفی می‌کنند که در آن ٥ مورد از نابسامانی‌های کد، مورد بررسی و بازسازی قرار گرفته است [١٧]. از بین ٥ مورد انتخابی، نابسامانی‌های Viewholder و Wakelock در این مقاله نیز مورد بررسی قرار گرفته‌اند. بعد از انجام بازسازی‌ها، نویسندگان نتیجه‌گیری کرده‌اند که انجام بازسازی‌های کد انتخابی می‌تواند به بهود انرژی برنامه‌های موبایلی کمک کند. نکته قابل توجه اینجا است که ۳۲ درصد از ۱٤۹ برنامه انتخابی در این مقاله دارای حداقل یکی از نابسامانی‌های شناخته شده کد بوده‌اند که این موضوع می‌تواند اهمیت این حوزه پژوهشی را نیز نشان دهد.

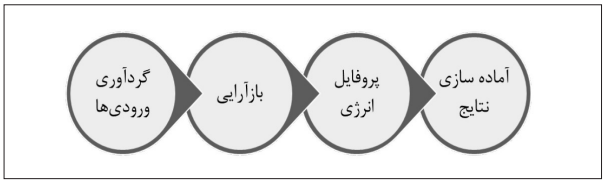
انور و همکاران هم با تمرکز بر ۲ نوع نابسامانی کد به نام‌های Duplicated code و Type checking نشان داده‌اند که بازسازی کد این نابسامانی‌ها می‌تواند تا ۱۰,۸ درصد به کاهش مصرف انرژی برنامه‌های موبایلی کمک کند[١٨]. همچنین، از نقاط قوت این پژوهش این است که

نشـــان داده‌اند که با وجود بازسازی کد انجام شده اگرچه مصرف انـــرژی کاهش پیدا می‌کند اما زمان اجرای برنامه دچار تغییر قابل ملاحظه‌ای نخواهد شـــد. اگرچه که تعداد محدود نابسامانی‌های انتخاب شده می‌تواند از نقاط ضعف این پژوهش شمرده شود.

پالومبا و همکاران تحقیق خود را بر روی اعمال همزمان روش‌های بازســـازی خودکار قـــرار داده‌اند[۱۹]. نقدی که نویسندگان بر پژوهش‌های کنونی وارد می‌دانند این است که اغلب این کارها، بازســـازی‌ها را به‌صورت مســـتقل در نظـــر گرفته و میزان تاثیر آن‌ها را مســـتقلاً مورد ارزیابی قـــرار می‌دهند. بنابراین، در این پژوهش طی مجموعه‌ای از ارزیابی‌ها، ترتیب و توالی انجام بازسازی‌ها مورد بررسی قرار گرفته است. نشـــان داده شده است که از نظر آماری می‌توان نشان داد که اعمال مستقل بازسازی‌ها بر کاهش مصرف انرژی تاثیرگذار بـــوده اما میزان این تاثیرگذاری متفاوت اســـت. همچنین، نشان داده شـــده است که اعمال همزمان بازســـازی‌ها می‌تواند به بهبود بیشـــتر مصرف انرژی در اکثر ســـناریوهای ارزیابی بررسی شده در این مقاله منتهی شـــود. اگرچه نتایج گزارش شـــده در این کار ارزشـــمند هستند اما تعداد نابســـامانی‌های مورد بررسی گرفتـــه در این مقاله و همین‌طور دامنه های کاربرد آزمون شده محدود است که همین موضوع، میزان اتکاپذیری نتایج را کم می‌کند.

٤- روش پیشنهادی

امروزه با توجه به محدود بودن منابع ذخیره انرژی در دستگاه‌های همراه مورد استفاده مانند گوشی تلفن همراه یکی از عوامل مهم در توسعه نرم‌افزارهای موبایلی میزان انرژی مصرفی اســـت. یکی از عواملـــی که می‌تواند میزان انرژی مصرفی توســـط نرم‌افزار را تحـــت تاثیر قرار دهد اســـتفاده از الگوها و نابسامانی‌های کد است. در این بخش به معرفی روشی برای بررسی تاثیر چند نابسامانی کد بر برنامه اندرویدی می‌پردازیم.



شکل ۱۱: نمودار پیشبرد فرآیندها در روش پیشنهادی

٤-۱ پیش‌برد فرآیندها

برای پیش‌برد فرآیندهای لازم در این مقاله طبق روندی که در نمای کلی شکل ۱۱ آمده است عمل می‌کنیم. در ادامه هر کدام از مراحل را به تفصیل شرح می‌دهیم.

٤-۲ گردآوری ورودی‌ها

در ایـــن مرحله به گـــردآوری ورودی‌هـــای آزمایش می‌پردازیم. ورودی‌های مورد نیاز شامل ۸ نابسامانی کد و ۲ برنامه اندرویدی اســـت. برای انتخاب برنامه‌ها فرآیند خاصی طی می‌شـــود تا بهترین و درســـت‌ترین انتخاب‌ها انجام شود. در ادامه، روش انتخاب را شرح می‌دهیم.

۴-۲-۱ برنامه‌ها

بـــرای انتخـــاب برنامه‌ها از F-droid، که یک فهرســـت از منابـــع برنامه‌های اندرویدی اســـت اســـتفاده می‌کنیم. این فهرســـت، رایگان و به‌صورت منبع باز اســـت. در حال حاضر، F-droid بیش از ۲۳۰۰ برنامه منبع باز ارائه می‌دهد. برنامه‌ها بر اساس معیارهای زیر انتخاب می‌شوند:

۱. جامـــع بودن: جامع بودن به این معنا اســـت که شـــامل تمامی موارد نابسامانی‌های کد مورد بررسی باشد.

۲. عدم استفاده از شبکه: تا بتوان تاثیر عامل اینترنت را از بین برد. به این دلیل که پهنای باند اینترنت از کنترل خارج است.

۳. منبـــع بـــاز: تا بتـــوان بـــر روی آن عملیات بازســـازی نابسامانی‌های کد را انجام داد.

٤. امتیاز Google Play: در قدم نهایی برای انتخاب دو منبع مورد بررســـی از بین چند برنامه انتخاب شـــده از معیار امتیاز و میزان نصب برنامه که در فروشگاه Google Play قابل دریافت است استفاده می‌گردد.

برنامه‌ها طبق فرآیند بالا که در شـــکل ۱۲ نشـــان داده

شکل ۱۲: فرآیند کلی انتخاب برنامه‌ها.

Algorithm name: refactor draw allocation
Inputs: View class
Outputs: refactored View class
foreach *onDraw* method in *View class* {
  if method block contains *object instantiation* {
    move *object instantiation* to class level
  }
}

الگوریتم ۱: بازسازی DrawAllocation

شـــده اســـت غربال می‌شـــوند تا در نهایت بـــه ۲ برنامه اندرویدی برسیم.

۴-۲-۲ نابسامانی‌های کد

لیست نابسامانی‌های کد مورد بررسی به شرح زیر است:

- Draw Allocation –
- Wake Lock –
- استفاده از HashMap –
- عدم استفاده از ViewHolder –
- Member Ignoring Method –
- عدم استفاده از recycle () –
- Obsolete layout params –
- Overdraw –

٤-۳ بازسازی

در این مرحله، سوژه‌های تعیین شده را تولید می‌کنیم. برای تولید هر ســـوژه باید نابسامانی‌های کد آن سوژه را بازســـازی نماییم. در این راســـتا ابتدا روش بازسازی هر نابسامانی کد را شرح می‌دهیم.

۴-۳-۱ Draw Allocation

در قطعـــه کد شـــکل ۱۳، رخداد این نابســـامانی کد را می‌بینیم که در متد onDraw یک شیء ساخته شده است.

برای بازسازی نابسامانی کد بالا از الگوریتم ۱ استفاده می‌کنیم:

طبق این الگوریتم، قطعه کد مربوط به ساخت شیء به

```
1  public class MyView extends View {
2      @Override
3      protected void onDraw(Canvas canvas) {
4          RectF rectF1 = new RectF();
5          ...
6          }
7  }
```

شکل ۱۳: نمونه کد Draw Allocation

```
1  public class MyView extends View {
2      RectF rectF1 = new RectF();
3      @Override
4      protected void onDraw(Canvas canvas) {
5      ...
6      }
7  }
```

شکل ۱۴: نمونه کد بازسازی شده DrawAllocation

```
1  public class MyFragment extends Fragment {
2      PowerManager.WakeLock wakeLock;
3      public void onClick(View view) {
4          wakelock.acquire();
5      }
6
7      @Override()
8      public void onPause() { super.onPause(); }
9  }
```

شکل ۱۵: نمونه کد WakeLock

بیرون از متد onDraw منتقل شده است. با توجه به این که این متد بارها فراخوانی می‌شود با این روش از فرایندهای اضافی تخصیص حافظه به شـــیء جلوگیری می‌کنیم. در قطعه کد شکل ١٤ نابسامانی کد بازسازی شده را می‌بینیم:

۴-۳-۲-4WakeLock ، قطعه کد شکل ۱۵ یک WakeLock را نشان می‌دهد که آزاد نشده است.

برای بازسازی نابسامانی کد بالا از الگوریتم ۲ استفاده می‌کنیم:

طبق الگوریتم بالا برای بازســـازی این نابســـامانی کد بایـــد هر نمونـــه‌ای از WakeLock را که در ردهی از جنس FragmentActivity است در روش onPause آزاد کنیم. به این ترتیب از WakeLock های بی‌مورد جلوگیری می‌کنیم. در قطعه کد شـــکل ١٦ نابســـامانی کد بازســـازی شده را می‌بینیم:

۴-۳-۳ استفاده از HashMap

قطعه کد شـــکل ۱۷ یک نمونه استفاده از HashMap را نشان می‌دهد.

Algorithm name: refactor wake lock

Inputs: FragmentActivity class

Outputs: refactored FragmentActivity class

foreach *WakeLock* instance in *FragmentActivity class* {

    call *release* method for *WakeLock* instance in *onPause* method

}

الگوریتم ۲: بازسازی WakeLock.

```
public class MyFragment extends Fragment {
    PowerManager.WakeLock wakeLock;
    public void onClick(View view) {
        wakelock.acquire();
    }

    @Override
    public void onPause() {
        super.onPause();
        if (wakeLock.isHeld()) wakeLock.release();
    }
}
```

شکل ۱۶: نمونه کد بازسازی شده WakeLock.

```
public class MyClass {
    ...

    public void method(){
    HashMap<String, Integer> map = new HashMap<>();
    ...
    }

    ...
}
```

شکل ۱۷: نمونه کد HashMap.

Algorithm name: refactor HashMap usage

Inputs: Class

Outputs: refactored Class

foreach *HashMap* instance in *Class* {

    change instance type to *ArrayMap*

}

الگوریتم ۳: بازسازی HashMap

```
public class MyClass {
    ...

    public void method(){
    ArrayMap<String, Integer> map = new ArrayMap<>();
    ...
    }

    ...
}
```

شکل ۱۸: نمونه کد بازسازی شده HashMap

```
...
@Override
public View getView (
    final int position,
    View convertView,
    ViewGroup parent
) {
    convertView = LayoutInflater.from(getContext())
        .inflate(R.layout.subforsublist, parent, false);

    TextView textView = ((TextView) convertView.findViewById(R.id.name));
    ...
}
```

شکل ۱۹: نمونه کد ViewHolder

برای بازسازی نابسامانی کد بالا از الگوریتم ۳ استفاده می‌کنیم:

طبـــق این الگوریتم، نمونه‌هایی از نوع HashMap را به ArrayMap تغییـــر می‌دهیم. از آنجایـــی که ArrayMap از لحاظ اســـتفاده از حافظه عملکرد بهتری دارد پس به بهینه کردن مصرف انرژی کمک می‌کند. در قطعه کد شـــکل ۱۸ نابسامانی کد بازسازی شده را می‌بینیم:

۴-۳-۴ عدم استفاده از ViewHolder

در قطعه کد شکل ۱۹ نمونه نابسامانی کد را می‌بینیم:

برای بازسازی نابسامانی کد بالا از الگوریتم ٤ استفاده می‌کنیـــم. با اســـتفاده از ایـــن الگوریتـــم از فراخوانی‌های اضافی تابـــع findViewById و همچنین ســـاختن شـــیء convertedView جلوگیری می‌شـــود. با توجه به هزینه‌بر بودن فراخوانی این توابع، بازســـازی آن بر بهینه‌ســـازی مصرف انرژی تاثیرگذار خواهد بود. در قطعه کد شکل ۲۰ نابسامانی کد بازسازی شده را می‌بینیم.

۴-۳-۵ Member Ignoring Method

برای بازسازی این نابسامانی کد از الگوریتم ٥ استفاده می‌کنیم. با استفاده از این الگوریتم توابعی از رده را که از متغیرها و بقیه توابع غیر static آن رده اســـتفاده نمی‌کنند static می‌کنیم. با توجـــه به این که توابع static در فضای جدایی از حافظه و مســـتقل از اشیاء دیگر ذخیره می‌شوند فارغ از این‌که چند شیء از آن رده داشته باشیم تابع static تنهـــا یک بار در حافظه ذخیره می‌شـــود. بدیـــن ترتیب از حافظه استفاده بهینه می‌کنیم.

```
Algorithm name: refactor not using ViewHolder
Inputs: Class
Outputs: refactored Class
create static class as ViewHolder
foreach view in xml {
        create public variable in ViewHolder class
        create public variable in Class
}
foreach getView method in Class {
        create variable of type ViewHolder with null value as viewHolderItem
        if convertedView is not initialized {
                initialize convertedView
                initialize viewHolderItem
                foreach view in xml {
                        call findViewById with id as argument
                        assign found view to matched variable of viewHolderItem
                }
                call setTag method for convertedView with viewHolderItem as argument
        } else {
                call getTag method for convertedView
                convert result type to ViewHolder
                assign result to viewHolderItem
        }
        foreach view variable in Class {
                assign matched variable value from viewHolderItem
        }
}
```

الگوریتم ۴: بازسازی عدم استفاده از ViewHolder

```
1  ...
2  private static class ViewHolderItem {
3      private TextView textView;
4  }
5
6  @Override
7  public View getView (
8      final int position,
9      View convertView,
10     ViewGroup parent
11 ) {
12     ViewHolderItem viewHolderItem;
13     if (convertView == null) {
14         convertView = LayoutInflater.from(getContext())
15             .inflate(R.layout.subforsublist, parent, false);
16         viewHolderItem = new ViewHolderItem();
17         viewHolderItem.textView = ((TextView) convertView.findViewById(R.id.name));
18         convertView.setTag(viewHolderItem);
19     } else {
20         viewHolderItem = (ViewHolderItem) convertView.getTag();
21     }
22     final TextView textView = viewHolderItem.textView;
23     ...
24 }
25 ...
```

شکل ۲۰: نمونه کد بازسازی شده ViewHolder

۴-۳-۶ عدم استفاده از () Recycle

```
Algorithm name: refactor Member Ignoring Method
Inputs: Class
Outputs: refactored Class
foreach method in Class {
        if method does not contain non-static
        class fields and other non-static class methods {
                change method type to static
        }
```

الگوریتم ۵: بازسازی MIM

```
1  public void method (AttributeSet attrs, int defStyle) {
2      final TypedArray array = getContext().obtainStyledAttributes(
3              attrs,
4              new int [] { 0 },
5              defStyle,
6              0
7          );
8
9      String example = array.getString(0);
10 }
```

شکل ۲۱: نمونه کد recycle

```
Algorithm name: refactor not using recycle()
Inputs: Class
Outputs: refactored Class
foreach method in Class {
        if method contains TypedArray instance {
                call recycle method for instance
        }
}
```

الگوریتم ۶: بازسازی Recycle

در قطعه کد شکل ۲۱ نمونه نابسامانی کد را می‌بینیم:

برای بازسازی این نابسامانی کد از الگوریتم ٦ استفاده می‌کنیم. طبق این الگوریتم بعد از استفاده از TypedArray آن را با اســتفاده از فراخوانی تابع recycle آزاد می‌کنیم. با توجه به این که این نوع با اســتفاده از منابع singleton پیاده‌ســازی می‌شــوند با فراخوانی تابع recycle فضای اشغالی حافظه آزاد می‌شود.

در قطعه کد شــکل ۲۲ نابسامانی کد بازسازی شده را می‌بینیم.

۴-۳-۷-4Obsolete layout params

در قطعه کد شکل ۲۳ نمونه نابسامانی کد را می‌بینیم.

برای بازسازی این نابسامانی کد از الگوریتم ۷ استفاده می‌کنیم:

```
< LinearLayout >
    <TextView android:id ="@+id/ name "
    android:layout_width =" wrap_content "
    android:layout_height =" wrap_content "
    android:layout_alignParentBottom =" true ">
    </ TextView >
</ LinearLayout >
```

شکل ۲۳: نمونه کد OLP

Algorithm name: refactor Obsolete layout params
Inputs: xml
Outputs: refactored xml
foreach *attribute* in *xml* {
    if *attribute* does not affect UI {
        remove attribute
    }
}

الگوریتم ۷: بازسازی OLP

```
< LinearLayout >
    <TextView android:id ="@+id/ name "
    android:layout_width =" wrap_content "
    android:layout_height =" wrap_content "
    </ TextView >
</ LinearLayout >
```

شکل ۲۴: نمونه کد بازسازی شده OLP

Algorithm name: refactor Overdraw
Inputs: View class
Outputs: refactored View class
if *View* class has covered all parent view {
    remove parent view background in onDraw method
}

الگوریتم ۸: بازسازی OverDraw

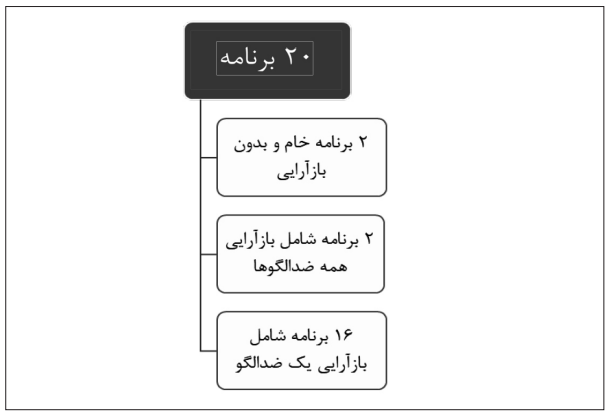
طبق این الگوریتم، در فایل‌های منبع اگر یک ویژگی در UI تاثیری نداشته باشد اضافی محسوب می‌شود و می‌توان آن را حذف نمود. در این مثال :alignParentBottom_android layout تاثیری بر LinearLayout ندارد و مورد استفاده آن اینجا نیست پس آن را حذف می‌کنیم. نتیجه در شکل ۲٤ نشان داده شده است.

۴-۳-۸-4Overdraw

برای بازسازی این نابسامانی کد از الگوریتم زیر استفاده می‌کنیم:

```
@Override
void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    if (hasCoverdParentView()) {
        getWindow().setBackgroundDrawable(null);
    }
}
```

شکل ۲۵: نمونه کد بازسازی شده OverDraw



شکل ۲۶: دسته‌بندی سوژه‌ها

طبق الگوریتم بالا اگر یک view تمام parent view خود را در بر بگیرد می‌توان در متد onDraw آن زمینه parent view را حذف کرد. بدین ترتیب از این‌که یک پیکسل بارها ترسیم شود جلوگیری می‌کنیم. با توجه به این که ترسیم پیکسل فرایند هزینه‌بری است این بازسازی تاثیر خوبی بر بهینه کردن مصرف انرژی دارد.

٤-٤ تولید سوژه‌ها

حال با توجه به این که روش بازسازی هریک از نابسامانی‌های کد را می‌دانیم به تولید سوژه‌های مورد بررسی می‌پردازیم.

همان‌طور که در شکل ۲٦ نشان داده شده است، سوژه‌های ما شامل ۲۰ برنامه است. سه دسته سوژه داریم که در ادامه به شرح هر یک می‌پردازیم.

۴-۴-۱ سوژه‌هایی بدون بازسازی

در این دسته به تعداد برنامه‌ها یک نسخه خواهیم داشت به‌طوری که هیچ تغییر و بازسازی‌ای در آن ایجاد نمی‌کنیم. با توجه به این که ۲ برنامه مورد بررسی قرار می‌گیرد

درنتیجه تعداد سوژه‌های این دسته برابر با ۲ است.

۴-۴-۲ سوژه‌هایی با بازسازی یک نابسامانی کد

در این دسته به تعداد نابسامانی‌های کد برای هر برنامه یک نسخه خواهیم داشت به‌طوری که در هر نسخه تنها یک نابسامانی کد بازسازی شده است. هر نسخه یک سوژه مورد بررسی خواهد بود. یعنی داریم:

$$\text{تعداد سوژه‌ها} = (\text{تعداد برنامه‌ها}) \times (\text{تعداد کد نابسامانی‌های}) \quad (۷)$$

درنتیجه تعداد سوژه‌های این دسته برابر با ۱۶ است.

۴-۴-۳ سوژه‌هایی با بازسازی همه نابسامانی‌های کد

در این دسته به تعداد برنامه‌ها یک نسخه خواهیم داشت. به‌طوری که در هر نسخه همه نابسامانی‌های کد بازسازی شده‌اند. هر نسخه یک سوژه مورد بررسی است. درنتیجه تعداد سوژه‌های این دسته برابر با ۲ است.

۴-۵ پروفایل انرژی

در این مرحله باید سوژه‌های به‌دست آمده را مورد آزمایش قرار دهیم. این آزمایش‌ها شامل پروفایل کردن انرژی مصرفی هر سوژه می‌شوند. از این آزمایش‌ها داده‌های خروجی لازم را گردآوری می‌کنیم. محیط انجام این آزمایش‌ها و همچنین روش انجام آن‌ها در ادامه شرح داده می‌شود.

۴-۵-۱ محیط آزمایش

محیط انجام آزمایش یک دستگاه تلفن همراه خواهد بود. حال با توجه به این که هدف ما از انجام آزمایش گردآوری اطلاعات دقیق مربوط به پروفایل انرژی مصرفی سوژه‌ها است و از آنجایی که برنامه‌ها و فرایندهای مختلفی در یک دستگاه تلفن همراه در حال فعالیت هستند پس باید اطمینان حاصل کنیم که این فعالیت‌های اضافی کمترین تاثیر ممکن را بر اعداد به‌دست آمده دارد. برای رسیدن به این مهم چند نکته را باید رعایت کنیم. این نکته‌ها را در ادامه شرح می‌دهیم.

۱. **حذف برنامه‌ها و فرایندهای اضافی**: در این گام تمام برنامه‌ها و تمام فرایندهای اضافی را حذف می‌کنیم. بدین ترتیب تاثیر آن‌ها را بر داده‌های حاصل از پروفایل انرژی سوژه‌ها جلوگیری می‌کنیم.

۲. **قطع ارتباط‌ها**: در این گام تمام ارتباط‌های اضافی را قطع می‌کنیم. بدین ترتیب تاثیر آن‌ها را بر داده‌های حاصل از پروفایل انرژی سوژه‌ها جلوگیری می‌کنیم. این ارتباط‌ها شامل موارد زیر می‌شوند:۱. ارتباط شبکه۲، ارتباط بلوتوث و ۳. ارتباط اینترنت.

۳. **شرایط دستگاه**: برای این‌که عوامل موجود در خود دستگاه را کنترل کنیم و از تاثیر آن‌ها بر نتیجه آزمایش‌ها جلوگیری کنیم موارد زیر را انجام خواهیم داد: ۱. کم کردن نور صفحه دستگاه تا حد امکان و ۲. شارژ کامل باتری دستگاه قبل از هر بار آزمایش.

برای پروفایل کردن انرژی سوژه‌ها از ابزار Battery Historian استفاده می‌کنیم. این ابزار میزان مصرفی انرژی برنامه را برای هر آزمایش بر حسب ژول[10] به ما می‌دهد. برای آزمایش هر سوژه یک بار فرایند زیر را طی می‌کنیم. در ادامه هر گام را شرح می‌دهیم.

۱. **نصب سوژه**: در این گام سوژه مورد نظر را بر روی دستگاه نصب می‌کنیم.

۲. **آماده سازی شرایط آزمایش**: در این گام شرایط دستگاه را آماده اجرای گام بعدی می‌کنیم. این آماده‌سازی شامل مواردی است که در بخش قبلی مورد بحث قرار گرفت.

۳. **اجرای سناریو**: در این گام سناریوی طراحی شده را روی سوژه نصب شده اجرا می‌کنیم. به ازای هر برنامه مورد بررسی که در این مقاله برابر با ۲ است یک سناریوی تست طراحی می‌شود این سناریو طوری طراحی می‌شود که تمام بخش‌های برنامه را شامل شود. هر سناریو در هر آزمایش ۲۰ بار تکرار می‌شود. این تعداد تکرار باعث می‌شود تا مقادیر مصرفی انرژی بزرگ‌تر شوند و در بخش مقایسه قابل استنادتر باشند.

۴. **ذخیره داده‌ها**: در گام آخر داده‌های مربوط به

10- Jules

پروفایل انرژی سوژه‌ها را که شامل مقدار انرژی مصرفی بر حسب ژول است ذخیره می‌کنیم.

٤-۶ آماده‌سازی نتایج

در این قسمت، داده‌های ذخیره شده در گام قبل را آماده می‌کنیم. ابزار نمایه‌گر، میزان انرژی مصرف شده را بر حسب ژول به ما می‌دهد. حال ما باید میزان تغییر انرژی مصرفی را به ازای هر بازسازی به‌دست آوریم. پس داریم:

$$\Delta E = \frac{E_2 - E_1}{E_1} \times 100 \qquad (۸)$$

در تساوی بالا E2 برابر با انرژی اندازه‌گیری شده پس از بازسازی بوده و E1 برابر با انرژی اندازه‌گیری شده قبل از بازسازی است. بدین ترتیب نتیجه نهایی برابر است با درصد میزان تغییر مصرف انرژی پس از بازسازی. در این بخش سعی کردیم با توجه به نتایجی که از بخش پیشین یعنی بخش بررسی کارهای مرتبط گرفتیم و مزایا و معایبی که از کارهای کروز و دیگران و کوتو و دیگران به‌دست آوردیم روشی را ارائه دهیم تا هم معایب روش‌های قبلی را بپوشاند و هم مزایای آن‌ها را بهبود بخشد. همچنین برای ارزیابی و بررسی نتایج روشی ارائه شد تا بتوانیم از صحت نتایج به‌دست آمده مطمئن شویم. همچنین نتایج به‌دست آمده را از دو منظر می‌توان بررسی کرد. یکی این که میزان تغییر انرژی مصرفی بعد از بازسازی همه نابسامانی‌های کد را بررسی می‌کنیم. نتیجه این بررسی به ما نشان خواهد داد که آیا میزان تغییر انرژی مصرفی در صورتی که کاهشی باشد ارزش صرف زمان و منابع برای بازسازی این نابسامانی‌های کد را دارد یا خیر. دیگر این‌که همه نابسامانی‌های کد را بر اساس درصد تغییر مصرف انرژی از مثبت به منفی مرتب می‌کنیم به این ترتیب لیستی از نابسامانی‌های کد اولویت‌بندی شده داریم. در صورتی که تولیدکننده‌ای با زمان و منابع محدود بخواهد میزان مصرف انرژی برنامه خود را کاهش دهد می‌تواند مطابق این لیست اولویت‌بندی شده نابسامانی‌های کد را در برنامه خود بیابد و آن‌ها را بازسازی کند و بدین ترتیب بیشترین میزان صرفه‌جویی در مصرف انرژی را خواهد داشت.

۵- ارزیابی روش پیشنهادی

در بخش قبل فرایند روش پیشنهادی و الگوریتم‌های پیشنهادی برای بازسازی نابسامانی‌های کد شرح داده شد. در این بخش ابتدا به بررسی و ارزیابی آزمایش‌ها و در ادامه با مقایسه نتایج انرژی مصرفی سوژه‌ها با روش‌های دیگر به ارزیابی روش شرح داده شده می‌پردازیم. با توجه به این که روش‌های دیگر روی برنامه‌های دیگری و در شرایط متفاوتی آزمایش شده‌اند، برای این‌که مقایسه درستی انجام گیرد با استفاده از روش‌های آن‌ها سوژه‌هایی مشابه با سوژه‌های این مقاله را تولید می‌کنیم و در شرایط مشابه میزان انرژی مصرفی آن‌ها را اندازه می‌گیریم.

۵-۱ محیط آزمایش

برای انجام آزمایش‌های مورد نیاز از یک گوشی اندرویدی استفاده می‌کنیم که دارای مشخصه‌های زیر است:

۱- مدل: Huawei Honor 8

۲- حافظه: ٤ گیگابایت

۳- اندروید: ۷٫۰

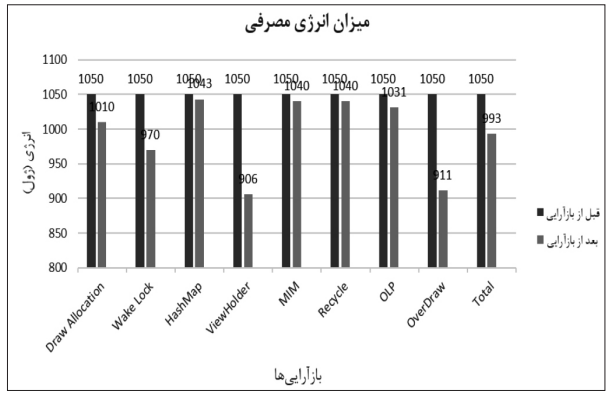
٤- حافظه داخلی : ۳۲ گیگابایت.

٥- مدل CPU: Hisilicon Kirin 950

این گوشی را مطابق روش گفته شده در فصل پیشین آماده می‌کنیم و از آن برای نصب نسخه‌های آزمایشی برنامه‌ها و اجرای سناریوهای طراحی شده و در نهایت برای پروفایل انرژی مصرفی برنامه در انتهای هر آزمایش استفاده می‌کنیم.

۵-۲ نحوه تولید داده‌ها

برنامه‌هایی که در این پژوهش استفاده شده‌اند مطابق با روشی که در فصل پیشین گفته شده انتخاب شده‌اند. این دو برنامه به زبان جاوا نوشته شده‌اند و ما برای



شکل ۲۷: نمودار میزان مصرف انرژی روش پیشنهادی.

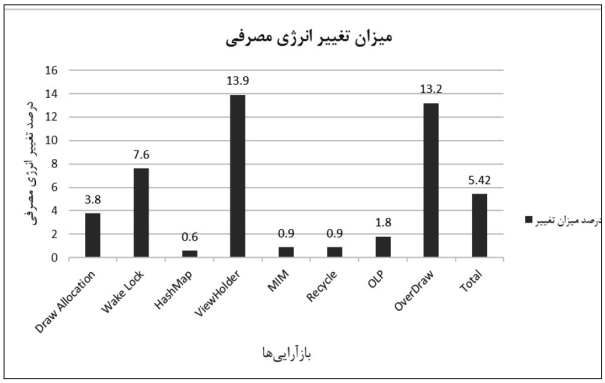
بازسازی نابسامانی‌های کد و تولید سوژه‌ها از ابزار اندروید استودیو نسخه ٤,٠ استفاده می‌کنیم. بعد از آماده کردن محیط آزمایش یکی از سوژه‌ها را بر روی گوشی نصب می‌کنیم. مطابق روش شرح داده شده در فصل پیشین آزمایش‌ها را اجرا می‌کنیم. می‌دانیم که برای هر سوژه ۲۰ بار یک آزمایش تکرار می‌شود. انرژی مصرفی اندازه‌گیری شده در پایان هر آزمایش را در یک فایل اکسل یادداشت می‌کنیم. در انتها انرژی مصرفی هر سوژه را با میانگین گرفتن از نتیجه آزمایش‌های مکرر انجام شده تعیین می‌کنیم.

۵-۳ ارزیابی نتایج

در این قسمت نتایج کلی آزمایش‌ها را ارزیابی می‌کنیم و بررسی می‌کنیم که آیا روش پیشنهادی به‌طور کلی مفید و کارا هست یا خیر. برای رسیدن به جواب این پرسش نتایج به‌دست آمده را از دو بعد بررسی می‌کنیم که در ادامه به شرح آن‌ها می‌پردازیم.

۵-۳-۲ میزان انرژی مصرفی

در این قسمت، میزان تغییر انرژی مصرفی را بر حسب ژول بررسی می‌کنیم. طبق نمودار شکل ۲۷، همه بازسازی‌ها تاثیر مثبتی بر مصرف انرژی داشته‌اند به این معنی که انجام هر یک از این بازسازی‌ها مصرف انرژی را کاهش می‌دهد. همچنین، می‌بینیم که اگر همه بازسازی‌ها انجام شوند در مجموع تاثیر قابل توجهی بر میزان مصرف انرژی می‌گذارد.



شکل ۲۸: نمودار میزان تغییر مصرف انرژی روش پیشنهادی

۵-۳-۲ میزان تغییر انرژی مصرفی

در این قسمت، با استفاده از فرمولی که در فصل پیش معرفی شد درصد میزان تغییر انرژی را به‌دست می‌آوریم و آن را بررسی می‌کنیم. همان‌طور که در نمودار شکل ۲۸ می‌بینیم، میزان تاثیر هر یک از بازسازی‌ها متفاوت با بقیه بوده و مشاهده می‌شود که بازسازی‌هایی مانند OverDraw و ViewHolder که مربوط به عملیات ترسیم پیکسل‌ها هستند میزان تغییر انرژی مصرفی آن‌ها بیشتر است.

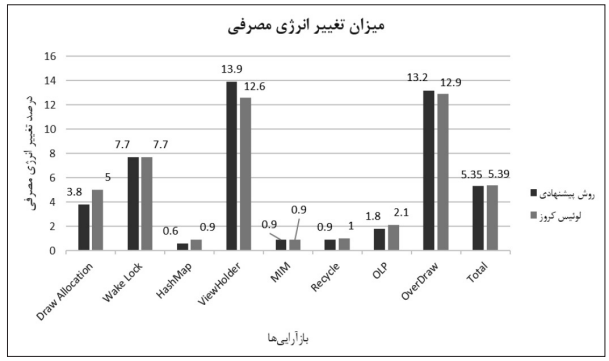
٤-۵ مقایسه با روش‌های دیگر

در این قسمت، نتایج انرژی مصرفی روش پیشنهادی را با نتایج انرژی مصرفی روش‌های دیگر مقایسه می‌کنیم. این روش‌های دیگر شامل روش کروز و دیگران و روش کوتو و دیگران هستند. ابتدا روش پیشنهادی را با هر یک از این دو روش مقایسه می‌کنیم و در ادامه یک مقایسه کلی انجام می‌دهیم.

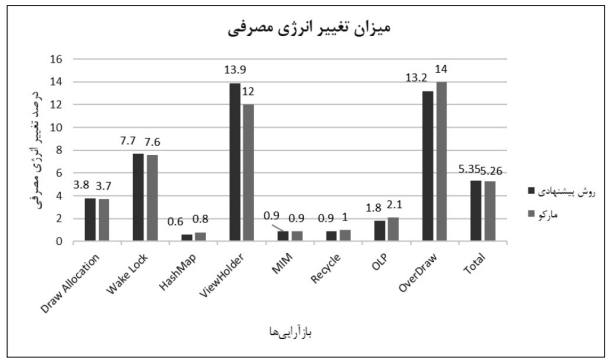
۵-۴-۱ مقایسه با روش کروز و دیگران

در این قسمت، درصد میزان تغییرات انرژی روش پیشنهادی را با روش کروز و دیگران مقایسه می‌کنیم. همان‌طور که در نمودار شکل ۲۹ مشاهده می‌شود در بعضی بازسازی‌ها روش پیشنهادی عملکرد بهتری داشته ولی در بعضی دیگر عملکرد ضعیف‌تری داشته است. همچنین، در دو مورد عملکرد هردو یکسان بوده است.

روش پیشنهادی در ۲ مورد بهتر عمل کرده است:



شکل ۲۹: نمودار درصد میزان تغییر انرژی مصرفی روش پیشنهادی



شکل ۳۰: نمودار مقایسه میزان مصرف انرژی با روش کوتو و دیگران

۱- OverDraw: دلیل عملکرد بهتر در این زمینه را می‌توان این در نظر گرفت که در روش پیشنهادی روش hasCoveredParentView بهتر عمل کرده و viewهای بیشتری را شناسایی می‌کند.

۲- ViewHolder: دلیل عملکرد بهتر در این زمینه static قرار دادن رده‌های ViewHolder می‌باشد.

روش پیشنهادی در ٤ مورد عملکرد ضعیف‌تری داشته است:

۱- OLP: الگوریتم پیشنهادی تعداد کمتری از attributeهای xml را پیدا و حذف کرده‌است و درنتیجه عملکرد ضعیف‌تری داشته است.

۲- Recycle: در بعضی قسمت‌های کد که به TypedArray دسترسی می‌یابد الگوریتم ما نتوانسته آن را شناسایی و روش Recycle را برای آن فراخوانی کند.

۳- HashMap: دلیل عملکرد بهتر روش کروز و دیگران در این زمینه می‌تواند تعریف کردن متغیرها در سطح روش باشد البته در صورتی که امکان آن وجود داشته باشد.

٤- Draw Allocation: دلیل عملکرد بهتر در این زمینه را می‌توان این در نظر گرفت که در روش پیشنهادی متغیرها را در سطح رده تعریف می‌کنیم. اما در روش کروز و دیگران صرفاً در روش دیگری تعریف می‌شوند.

در ۲ مورد عملکرد هر دو یکسان بوده است: MIM، Wake Lock. همچنین این نمودار نشان می‌دهد که به‌طور کلی عملکرد روش کروز و دیگران از روش پیشنهادی بهتر بوده است.

۵-۴-۲ مقایسه با روش کوتو و دیگران

در این قسمت درصد میزان تغییرات انرژی روش پیشنهادی را با روش کوتو و دیگران مقایسه می‌کنیم. همان‌طور که در نمودار شکل ۳۰ مشاهده می‌شود در بعضی بازسازی‌ها روش پیشنهادی عملکرد بهتری داشته ولی در بعضی دیگر عملکرد ضعیف‌تری داشته است. همچنین در یک مورد عملکرد هردو یکسان بوده است.

روش پیشنهادی در ۳ مورد بهتر عمل کرده است:

۱- Draw Allocation: دلیل عملکرد بهتر در این زمینه را می‌توان این در نظر گرفت که در روش کوتو و دیگران، متغیرها در سطح رده تعریف نمی‌شود.

۲- ViewHolder: دلیل عملکرد بهتر در این زمینه static قرار دادن رده‌های ViewHolder در روش پیشنهادی است.

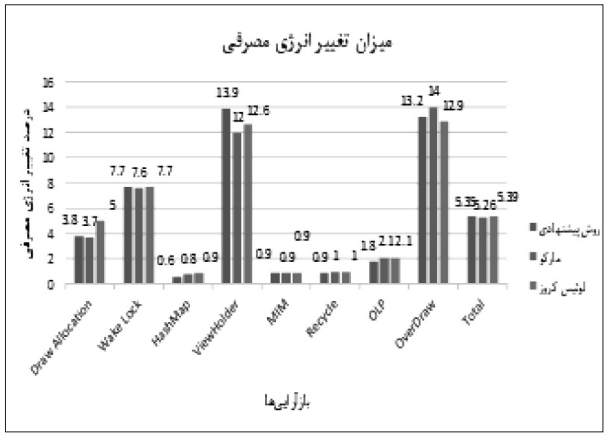
۳- Wake Lock: دلیل بهتر عمل کردن روش پیشنهادی پیدا کردن WakeLock ها هم در Activityها و هم در Fragmentها است.

روش پیشنهادی در ٤ مورد عملکرد ضعیف‌تری داشته است:

۱- OLP: الگوریتم پیشنهادی تعداد کمتری از attributeهای xml را پیدا و حذف کرده‌است و درنتیجه عملکرد ضعیف‌تری داشته است.

۲- Recycle: در بعضی قسمت‌های کد که به TypedArray دسترسی می‌یابد الگوریتم ما نتوانسته آن را شناسایی و روش Recycle را برای آن فراخوانی کند.

۳- HashMap: دلیل عملکرد بهتر روش کوتو و دیگران



شکل ۳۱: نمودار مقایسه میزان مصرف انرژی با روش‌های مورد مقایسه

در ایـــن زمینـــه می‌تواند تعریف کردن متغیرها در ســـطح روش باشد.

٤- OverDraw: دلیـــل عملکـــرد بهتـــر در این زمینه را می‌توان این در نظر گرفت که در روش پیشـــنهادی روش hasCoveredParentView بهتـــر عمل کـــرده و viewهای بیشتری را شناسایی می‌کند.

در ۲ مورد عملکرد هر دو یکســـان بوده اســـت: MIM. همچنین این نمودار نشـــان می‌دهد که به‌طور کلی عملکرد روش پیشنهادی از روش کوتو و دیگران بهتر بوده است.

۵-۴-۳ مقایسه با روش کروز و دیگران و کوتو و دیگران

در نمودار شـــکل ۳۱، مقایســـه کلی بین ســـه روش را مشـــاهده می‌کنیم که جزئیات آن را در قســـمت‌های قبلی مورد بحث قرار دادیم.

در ایـــن بخش به مقایســـه روش‌های مشـــابه کوتو و دیگران و کروز و دیگران با روش پیشـــنهادی در بررسی میزان تاثیر بازسازی نابسامانی‌های کد در میزان مصرف انـــرژی برنامه‌های اندرویدی پرداختیـــم. همان‌طور که از نتایج ارزیابی برمی‌آید روش پیشـــنهادی در برخی موارد توانســـته اســـت بهتر از دو روش دیگر عمـــل کند و ما در این مـــوارد به اهداف خود دســـت یافته‌ایم. اما از ســـوی دیگر روش پیشنهادی در برخی موارد نتوانسته عیب‌های دو روش دیگـــر را برطـــرف کند و عملکرد بهتری داشـــته باشد.

۶- نتیجه‌گیری

در این مقاله روش‌هایی برای بازسازی نابسامانی‌های کد انتخاب شـــده ارائه کردیـــم و الگوریتم‌هایـــی را برای بازســـازی آن‌ها طراحی کردیم. همچنین، یک ســـازوکار برای بررسی میزان تاثیر این بازسازی‌های پیشنهاد شده طراحـــی کردیم. در ادامه، به بررســـی نقاط قوت و ضعف نتایج به‌دســـت آمده حاصل از روش پیشنهادی پرداختیم. همچنین نتایج خود را با روش‌های دیگر موجود مقایســـه کردیم تا میزان کارایی آن را بسنجیم. از سویی همان‌طور که از مقایســـه نتایج این روش با روش‌های دیگر دیدیم در بعضی نابســـامانی‌های کد توانستیم نقص‌های روش‌های قبلی را پوشـــش دهیم و عملکرد آن‌ها را بهبود ببخشـــیم. خصوصاً در مواردی که به عملیات ترسیم پیکسلی مرتبط بودند مشاهده شد که روش‌های پیشنهادی به‌طور متوسط عملکرد بهتری نسبت روش‌های دیگر داشتند.

از ســـوی دیگر مشـــاهده کردیم اعمال بازسازی‌های نابســـامانی‌های کد در فرآیند توسعه برنامه‌های موبایلی می‌تواند تاثیر مثبتی بر میزان مصرف انرژی گوشـــی‌های اندرویدی داشـــته باشـــد. در واقع برای تولیدکنندگانی که به دنبـــال کاهش میزان مصـــرف برنامه‌هـــای اندرویدی خود هســـتند یکی از راه‌هـــای مؤثر می‌تواند بازســـازی نابســـامانی‌های کد مطرح شـــده باشـــد. از مزایایی که در روش پیشـــنهادی وجود دارد می‌توان به این اشـــاره کرد که در بازســـازی برخی از نابسامانی‌های کد از روش‌های مشابه مورد بررسی عملکرد بهتری داشتیم. همین‌طور یکی دیگر از مزیت‌های مهم این پژوهش بهبود عیب‌ها در فرایند ارزیابی و ســـنجش انرژی برنامه‌های اندرویدی است. در این مقاله با استفاده از ابزاری خاص منظوره برای سنجش انرژی برنامه‌های اندرویـــدی به مراتب به نتایج دقیق‌تری دست یافتیم. همچنین از دیگر مزایای این پژوهش این است که نابســـامانی‌های کد را بر اســـاس میزان تاثیر آن‌ها بر مصرف انرژی برنامه‌های اندرویدی رتبه‌بندی کردیم.

از معایبی که در روش پیشـــنهادی وجود دارد می‌توان

به این اشـــاره کرد که در برخی از موارد که روشی برای بازســـازی نابســـامانی‌های کد ارائه کردیم با این که سعی بر این بود که با بررســـی دقیق روش‌های مشابه و تحلیل متدهای آنان ایرادهای آن‌ها را به‌دســـت آورده و در جهت رفع و بهبود آن‌ها تا حد امکان بکوشیم اما به این مهم دست نیافتیم. از معایب دیگر این پژوهش نبود سخت‌افزار مناسب جهت انجام آزمایش‌های بیشتر و دقیق‌تر بود. با این که در این پژوهش سعی شد از ابزارهای دقیقی استفاده شود اما وجود سختافزار مناسب می‌توانست نتایج را بیش از آنچه که اکنون هست معتبر و قابل استناد کند.

۷- کارهای آینده

همان‌طور که می‌دانیم نابســـامانی‌های کد بسیاری در بحث توسعه برنامه‌های اندرویدی وجود دارد و ما در این مقاله تنها به ۸ مورد از آن‌ها پرداختیم. همچنین در قسمت ارزیابی تنها بر روی دو برنامه اندرویدی نتایج را بررسی کردیم. بـــا توجه به این موارد می‌تـــوان این پژوهش را با ایده‌های زیر ادامه داد:

۱) در این پژوهش تنها به ۸ مورد از نابسامانی‌های کد شناخته شده در برنامه‌نویســـی اندروید پرداخته شد. اما همان‌طور که می‌دانیم نابســـامانی‌های کد بسیاری در این زمینه وجود دارند که تعداد زیادی از آن‌ها به بحث مصرف انرژی برنامه مرتبط هســـتند. بنابراین، یکی از کارهایی که می‌توان انجام داد بررســـی نابســـامانی‌های کد پرکاربرد دیگر در بحث کاهش مصرف انرژی است.

۲) در ایـــن پژوهـــش بـــا توجه بـــه منابـــع و امکانات، بازســـازی‌های پیشـــنهادی تنها بر روی تعـــداد محدودی از برنامه‌های اندرویدی بررســـی شـــد. با توجه به این که امروزه برنامه‌های پرتعداد و متنوعی تولید شـــده و مورد اســـتفاده قرار می‌گیرد یکی دیگـــر از کارهایی که می‌توان انجام داد بررسی روش ارائه شده بر روی طیف دیگری از برنامه‌های اندرویدی است.

۳) در ایـــن پژوهـــش برای هـــر برنامه و بـــرای انجام آزمایش روی هر یک از برنامه‌ها تعداد محدودی ســـناریو نوشـــته شـــد. با توجه با این که هر برنامه به شکل‌ها و در ســـناریوهای متنوعی می‌تواند مورد اســـتفاده قرار گیرد یکـــی دیگر از کارهایی کـــه می‌تواند انجام گیـــرد افزایش تعداد ســـناریوها برای برنامه‌های مورد بررســـی و تکرار آزمایش‌ها است.

٤) همان‌طور که می‌دانیم الگوهای طراحی زیادی وجود دارنـــد که تولیدکننـــدگان اندروید نیز از آن‌ها در توســـعه برنامه‌های اندرویدی خود اســـتفاده می‌کنند. یکی دیگر از کارهایی که می‌توان انجام داد بررسی این الگوهای طراحی پرکاربرد در برنامه‌نویســـی اندروید است که آیا از لحاظ مصرف انرژی مفید هستند یا خیر.

مراجع

[1] A. Carette, M. A. Ati Younes, G. Hecht, N. Moha and R. Rouvoy, “Investigating the energy impact of Android smells,” in Proceedings of the 2017 IEEE 24th International Conference on
Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria, pp. 115-126, 2017.

[2] R. Saborido, R. Morales, F. Khomh, Y. Guéhéneuc, and G. Antoniol, Getting the most from map data structures in Android, Empirical Software Engineering, vol. 23, no. 5, pp.2829–2864, 2018.

[3] M. Couto, J. Saraiva, J. Paulo Fernandes, “Energy Refactorings for Android in the Large and in the Wild”, in Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, United Kingdom, pp.55-62, 2020.

[4] L. Cruz and R. Abreu, “Performance-Based Guidelines for Energy-Efficient Mobile Applications,” in Proceedings of the 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), Pittsburgh, USA, pp. 46-57, 2017.

[5] Y. Hu, J. Yan, D. Yan, Q. Lu, and J. Yan, “Lightweight energy consumption analysis and prediction for Android applications,” Special Issue on TASE, Science of Computer Programming, pp. 132–147, 2018.

[6] D. Connolly Bree, M. Ó Cinnéide, “ Automated Refactoring for Energy-Aware Software”, in Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), Luxembourg, pp.113-124, 2021.

[7] R. Verdecchia, R. A. Saez, G. Procaccianti, and P. Lago, “Empirical Evaluation of the Energy Impact of Refactoring Code Smells,” in Proceedings of the 5th International Conference on Information and Communication Technology for Sus-

2014.
[14] M. Fowler, “Refactoring: improving the design of existing code”, Addison-Wesly Professional, ISBN: 978-0201485677, 1999.
[15] E. V. de Paulo Sobrinho, A. De Lucia, M. de Almeida Maia, “A systematic literature review on bad smells—5w’s: which, when, what, who, where”, IEEE Transactions on Software Engineering, vol. 118, no.4, 2018.
[16] E. Lannone et al., “Refactoring Android-specific Energy Smells: A Plugin for Android Studio”, in Proceedings of the 28th International Conference on Program Comprehension, Seoul, Republic of Korea, pp.451-455, 2020.
[17] L. Cruz, R. Abreu, “Using Automatic Refactoring to Improve Energy Efficiency of Android Apps”, in Proceedings of CIbSE XXI Ibero-American Conference on Software Engineering (CIbSE’18), Cordoba, Argentina, pp. 110-124, 2018.
[18] H. Anwar, D.Pfahl and S.N. Srirama, “Evaluating the Impact of Code Smell Refactoring on the Energy Consumption of Android Applications”, in Proceedings of the 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Kallithea, Greece, pp. 112-119, 2019.
[19] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, “Lightweight detection of Android-specific code smells Tshe aDoctor project,” in Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER), Hangzhou, China, pp. 487–491, 2020.
tainability, Toronto, Canada, pp. 365–383, 2018.
[8] G. Pinto, F. Francisco Soares-Neto, and F. Castor, “Refactoring for Energy Efficiency: A Reflection on the State of the Art,” in Proceedings of the 2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software, Florence, Italy, pp. 117-128, 2015.
[9] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia, On the impact of code smells on the energy consumption of mobile applications, Information and Software Technology, vol.112, no.5, ISSN 0950-5849, 2018.
[10] A. Snyder, “Encapsulation and inheritance in object-oriented programming languages”, in Proceedings of the Object-oriented Programming Systems, Languages and Applications Conference, Portland, USA, pp.3845, 1986.
[11] “Design Patterns”, URL: https://www.geeksforgeeks.org/design-patterns-set-1-introduction/, Access Date: 20 September 2019.
[12] M. Linares-V´asquez, G. Bavota, C. Bernal-C´ardenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, “Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study,” in Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, India, pp. 2–11, 2014.
[13] C. Sahin, L. Pollock, and J. Clause, “How do code refactorings affect energy usage?”, in Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM ‘14), Association for Computing Machinery, New York, NY, USA, Article 36, pp.1–10,