بیست و ششمین کنفرانس مهندسی برق ایران
۲۰ - ۱۸ اردیبهشت ۱۳۹۷
دانشگاه صنعتی نجاد

اجرای هدایت‌شده پویا-نمادین برنامک‌های اندرویدی[1] برای تولید خودکار ورودی آزمون

احسان عدالت[1]، محمود اقوامی‌پناه[2]، بابک صادقیان[3]
[1] دانشکده مهندسی کامپیوتر و فناوری اطلاعات، دانشگاه صنعتی امیرکبیر، تهران ، ehsan.e.71@aut.ac.ir
[2] دانشکده مهندسی کامپیوتر و فناوری اطلاعات، دانشگاه صنعتی امیرکبیر، تهران ، maghvami@aut.ac.ir
[3] دانشکده مهندسی کامپیوتر و فناوری اطلاعات، دانشگاه صنعتی امیرکبیر، تهران ، basadegh@aut.ac.ir

چکیده –اجرای پویا-نمادین روشی پویا برای آزمون نرم‌افزار است که می‌تواند به پوشش بالایی از کد دست یابد. مشکل این روش در برنامه‌های واقعی انفجار مسیر در اجرا است. بنابراین اجرای پویا-نمادین صرف برای برنامه‌های واقعی کارآمد نیست. از جمله نرم‌افزارهای محبوب، برنامک‌های اندرویدی هستند. آزمون برنامک‌های اندرویدی نسبت به برنامه‌های دیگر دارای چالش‌های جدید رخدادمحوربودن و وابستگی زیاد به *SDK*[2] است که سربار آزمون را بالا می‌برد. در این مقاله ما یک هیوریستیک را ارائه کرده‌ایم که اجرای پویا-نمادین را به صورت بهینه و هدایت‌شده روی برنامک‌های اندرویدی اعمال می‌کند. همچنین با تحلیل ایستا و استخراج گراف فراخوانی توابع همراه با پیمایش روبه‌عقب آن، نقطه شروع برنامک[3] را تولید می‌کنیم. با استفاده از گراف کنترل جریان بین تابعی و پیمایش روبه‌عقب آن نیز، اطلاعات مسیرهای دارای اولویت را در یک پشته ذخیره می‌نماییم. در این کار با ایده استفاده از کلاس‌های *Mock*، مسئله رخدادمحوربودن را حل کرده‌ایم. ضمن آنکه اجرای پویا-نمادین را با اطلاعات پشته مسیرهای مطلوب به صورت هدایت‌شده انجام می‌دهیم تا با محدود کردن فرایند آزمون به نقطه‌های شروع مشخص، سربار بالای آزمون برنامک‌ها را کاهش دهیم. برای ارزیابی راه‌کار ارائه‌شده، ابتدا ۱۰ برنامک دارای خطا را مطرح و پیاده‌سازی کردیم که ابزار ما تمامی خطاها را تشخیص داد. همچنین ۴ برنامک موردآزمون در ابزار *Sig-Droid* را با ابزار خود آزمودیم. نتایج نشان می‌دهد ابزار ما با سرعت بیشتری می‌تواند خطاهای برنامک را تشخیص دهد.

کلید واژه- اجرای پویا-نمادین، برنامک‌های اندرویدی، گراف فراخوانی توابع، گراف کنترل جریان بین تابعی، ورودی آزمون.

۱- مقدمه

اندروید محبوب‌ترین سیستم عامل حال حاضر گوشی‌های هوشمند است. با گسترش اندروید، توسعه برنامک‌های اندرویدی نیز رشد چشمگیری داشته‌اند به طوری که فقط در فروشگاه داخلی کافه بازار ، تاکنون بیش از ۱۴۰ هزار برنامک اندرویدی برای بیش از ۳۵ میلیون مخاطب داخلی منتشر شده است.[۱] با توجه به گسترش استفاده از برنامک‌های اندرویدی نیاز به یک سامانه خودکار برای آزمون این برنامک‌ها از سمت توسعه‌دهندگان، فروشگاه‌های اندروید و کاربران احساس می‌شود.

گوگل برای آسان کردن فرایند توسعه نرم‌افزار مجموعه‌ای از ابزار و کد یعنی SDK را ارائه داده است. برنامک‌های اندرویدی با اضافه کردن کدهای برنامه‌نویس به SDK تولید می‌شوند. این برنامک‌ها از جمله برنامه‌های رخدادمحور هستند که تفاوت عمده آنها با سایر برنامه‌ها، در هم‌تنیدگی زیاد با SDK است. این موضوع باعث می‌شود، برای اجرای یک قطعه کد ساده برنامه‌نویس، تعداد زیادی از قطعه کدهای SDK فراخوانی و اجرا شوند و این موضوع خودکار کردن فرایند آزمون برنامک‌ها را با چالش روبه‌رو می‌کند.

مهم ترین اهداف آزمون برنامک‌های اندرویدی، کشف خطا، استثنا و حالت خاتمه نامطمئن[4] است. هم اکنون برنامه نویسان اندروید برای یافتن خطای برنامک‌ها از نظرات کاربران و آزمون دستی برنامک استفاده می‌کنند. از آن جا که هزینه زمانی و تلاش نیروی انسانی برای آزمون دستی بسیار بالاست ما به ارائه یک روش نوین و کارا برای آزمون خودکار برنامک‌های اندرویدی پرداخته‌ایم.

پیش از این پژوهش کارهایی در حوزه آزمون برنامک‌های اندرویدی با سه رویکرد متفاوت انجام شده است. در رویکرد اول به طور بی‌قاعده[5] ورودی برنامک تولید می‌گردد. در ابزار Monkey [۲] به صورت دلخواه سعی می‌شود تا ورودی‌های

[1] Android Apps
[2] Software Development Kit
[3] Main Method
[4] Crash
[5] Random

آزمون برای برنامک تولید شود. مشکل اصلی این روش آن است که پوشش مناسبی از مسیرهای مختلف برنامک را نمی‌توان داشت. در رویکرد دوم ورودی برنامه به طور نظام‌مند[6] تولید می‌گردد. در ابزار Sig-Droid [۳] با استفاده از یک روش مشخص مانند اجرای نمادین به صورت جعبه‌سفید سعی می‌شود تا ورودی‌های برنامک تولید گردد. Sig-Droid تمام مسیرهای موجود در برنامک را به صورت نمادین اجرا می‌کند و همان طور که نویسنده بیان کرده است هدف آن پوشش هرچه بیشتر این مسیرها است. در رویکرد سوم مانند ابزار Swifthand [۴] مدلی از برنامه مانند مدل رابط گرافیکی کاربر از برنامک استنتاج می‌گردد و سپس بر اساس این مدل‌ها ورودی‌هایی برای برنامک تولید می‌شود که مسیرهای ناشناخته برنامک را بپیماید.

از میان روش‌های مختلف موجود در این حوزه ما روش پویا-نمادین را انتخاب کرده‌ایم که دارای پوشش قوی کد است. این روش برای اولین بار در [۵] ارائه شد. ابزار پیشنهادی ما با استفاده از اجرای پویا-نمادین به تولید خودکار ورودی آزمون برای برنامک‌های اندرویدی می‌پردازد. اجرای پویا-نمادین از جمله روش‌های رویکرد قاعده‌مند برای تولید ورودی آزمون برای برنامک‌های اندرویدی است.

در این کار، ابتدا ما apk برنامک را دیکامپایل می‌کنیم. سپس برای اینکه بتوانیم بر روی JVM[7] برنامک را با موتور SPF به شکل پویا-نمادین اجرا کنیم، لازم است تا نقطه شروع به برنامک را تولید نماییم. برای این منظور با تحلیل ایستا و استخراج گراف فراخوانی توابع و پیمایش روبه‌عقب آن، این تابع را تولید می‌کنیم.

در این پژوهش ما می‌خواهیم که اجرای پویا-نمادین برنامک به صورت هدفمند صورت گیرد تا بتوان خطاهای برنامک را سریع‌تر یافت. برای این هدف، از تحلیل ایستا و پیمایش روبه‌عقب گراف کنترل جریان بین تابعی بهره می‌بریم. در پیمایش روبه‌عقب، از عبارت رخداد خطا تا عبارت ریشه را می‌پیماییم و پشته شاخه‌های اولویت‌دار را مبتنی بر آن تولید می‌کنیم. برای حل مسئله رخدادمحوربودن و کامپایل‌شدن در JVM از کلاس‌های Mock به جای کلاس‌های SDK استفاده کردیم. کلاس Mock کلاسی است که تابع‌های کلاس اصلی را دارد، با این تفاوت که بدنه تابع حذف و یا به نحوی تغییر داده می‌شود که تنها، برنامک کامپایل شده و اجرای عادی داشته باشد بدون اینکه سربار کلاس‌های اصلی را داشته باشد.

با استفاده از توابع نقطه شروع برنامه، اجرای پویا-نمادین را محدود به تعدادی تابع خاص که موجب دستیابی به خطا می‌شوند، می‌کنیم. همچنین با استفاده از پشته شاخه‌های اولویت‌دار هیوریستیک خود را ارائه می‌دهیم که باعث می‌شود، به جای پیمایش عمق‌اول در اجرای پویا-نمادین، در هر دستور شرطی متناسب با اطلاعات پشته، شاخه بهینه که به خطا خواهد رسید را اجرا کنیم. این دو مورد موجب کاهش قابل توجه هزینه زمانی و سربار اجرای برنامک می‌شود. در نهایت با استفاده از ابزار Robolectric [۶] و ورودی‌های آزمون استخراج شده از اجرای پویا-نمادین، برنامک را اجرا می‌نماییم تا صحت عملکرد هیوریستیک و ورودی‌های تولید شده را بررسی کنیم.

به طور مختصر در این پژوهش دستاوردهای علمی زیر صورت گرفته است:

- اجرای برنامک روی JVM با استفاده از نقطه ورودی بدست‌آمده از تحلیل ایستا روی گراف فراخوانی توابع.
- اجرای هیوریستک پیشنهادی در SPF با تحلیل ایستا روی گراف کنترل جریان بین تابعی و به دست آوردن پشته شاخه‌های اولویت‌دار.
- اجرای پویا-نمادین برای برنامک‌های اندروید با استفاده از ایده کلاس‌های Mock نمادین و جلوگیری از مشکل انحراف مسیر و رخدادمحوربودن با استفاده از ایده ساخت Mock.
- ارائه یک هیوریستیک با ایده ترکیب تحلیل ایستا و پویا که موجب می‌شود، اجرای شاخه‌های دارای خطا را اولویت دهیم.

در این مقاله پس از مقدمه، در بخش دوم به پیش زمینه موضوع می‌پردازیم. در بخش پیش زمینه، اجرای پویا-نمادین و چالش‌های موجود برای اعمال آن در اندروید را مورد بررسی قرار داده‌ایم. سپس به بررسی کارهای پیشین در زمینه آزمون برنامک‌های اندرویدی و به طور خاص تولید ورودی‌های آزمون با استفاده از اجرای پویا-نمادین پرداخته‌ایم. در بخش سوم به طور مفصل به شرح روش پیشنهادی می‌پردازیم؛ در این بخش هیوریستیک انتخاب مسیر و موتور پویا-نمادین استفاده شده را شرح داده‌ایم. در بخش چهارم ارزیابی ابزار پیشنهادی را ارائه کرده‌ایم. در این بخش ابزار پیشنهادی را با ابزار آزمون Sig-Droid از نظر پوشش کد و زمان اجرا مقایسه کرده‌ایم. در بخش ششم نیز نتایج به دست آمده و دستاوردهای این پژوهش مورد بحث قرار گرفته‌اند.

۲- پیش زمینه

در این بخش اجرای پویا-نمادین را همراه با مثال و چالش‌های آزمون برنامک‌های اندرویدی را توضیح خواهیم داد.

[6] Systematic
[7] Java Virtual Machine

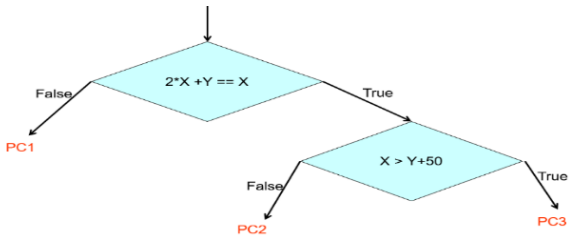
۲-۱- اجرای پویا-نمادین

اجرای پویا-نمادین از ترکیب دو اجرای عینی[8] و اجرای نمادین حاصل شده است. اجرای عینی به معنای اجرای عادی برنامه با مقادیر عینی به عنوان ورودی برنامه است. اما اجرای نمادین به معنی اجرای برنامه با در نظرگرفتن ورودی‌های برنامه به صورت نمادین است. در واقع در اجرای نمادین با درنظرگرفتن متغیرهای نمادین سعی می‌شود تا تمام شاخه‌های برنامه به صورت نمادین اجرا گردد و شرط مسیر مربوط به هر شاخه استخراج گردد. کد شکل ۱ را در نظر بگیرید. برای اجرای نمادین این تکه کد باید دو متغیر y,x را به صورت نمادین Y,X تعریف کرد. به شکل متناظر متغیر z هم مقدار نمادین 2X+Y را می‌پذیرد. با اجرای نمادین سه شرط مسیر PC1=not(X=2X+Y) و PC2=(X=2X+Y) و PC3=(X=2X+Y) ∧ (X>Y+50) استخراج می‌گردد که متناظر با سه شاخه مختلف از درخت اجرای شکل ۲ هستند. در واقع در هر مرحله شرط مسیر جدید با نقیض کردن شرط مسیر قبلی و بررسی امکان‌پذیر بودن آن توسط «حل‌کننده قید[9]» ساخته می‌شود. اما درصورتی که شرط مسیر، غیرخطی و پیچیده باشد، حل‌کننده قید قادر به حل آن نخواهد بود؛ بنابراین اجرای نمادین در آن نقطه با مشکل مواجه می‌گردد. اجرای پویا-نمادین با هدف برطرف کردن این مشکل، برنامه را به صورت همزمان با ورودی عینی و نمادین اجرا می‌نماید. از این رو زمانی که حل‌کننده قید قادر به حل شرط مسیر نباشد از مقادیر عینی برای ادامه اجرای برنامه استفاده می‌کند. این ایده موجب می‌شود تا اجرای پویا-نمادین پوشش کد قابل قبولی را ارائه دهد و تضمین کند که اغلب شاخه‌های قابل دسترس برنامه را پیموده است. در اجرای پویا-نمادین در هر مرحله حل‌کننده قید مقادیر ورودی عینی برای رسیدن به اجرای آن شاخه را در اختیار می‌گذارد.

```
1: public void test(int x ,int y){
2: int z=2x+y;
3: if(z==x){  //PC2
4:         if(x > y+50) //PC3
5:         }//PC1
6:}
```

شکل ۱: تکه کدی ساده

ما در این پژوهش موتور اجرای پویا-نمادین SPF [۷] را گسترش داده‌ایم به نحوی که قابلیت اجرای پویا-نمادین برای برنامک‌های اندرویدی را داشته باشد. موتور SPF در واقع مبتنی بر چارچوب کلی JPF [۸] که یک وارسی‌کننده مدل برای بایت‌کد جاوا است نوشته شده است. در SPF بایت‌کد برنامه به یک کد سه-آدرسه میانی تبدیل می‌شود. سپس این کد میانی بر روی یک ماشین مجازی تغییر یافته JVM اجرا می‌شود. همچنین این ابزار از تعداد زیادی از حل‌کننده‌های قید پشتیبانی می‌کند که با استفاده از آنها می‌توان قیدهای مختلف از جمله رشته‌ها را تحلیل نمود.



شکل۲: درخت اجرای شکل ۱.

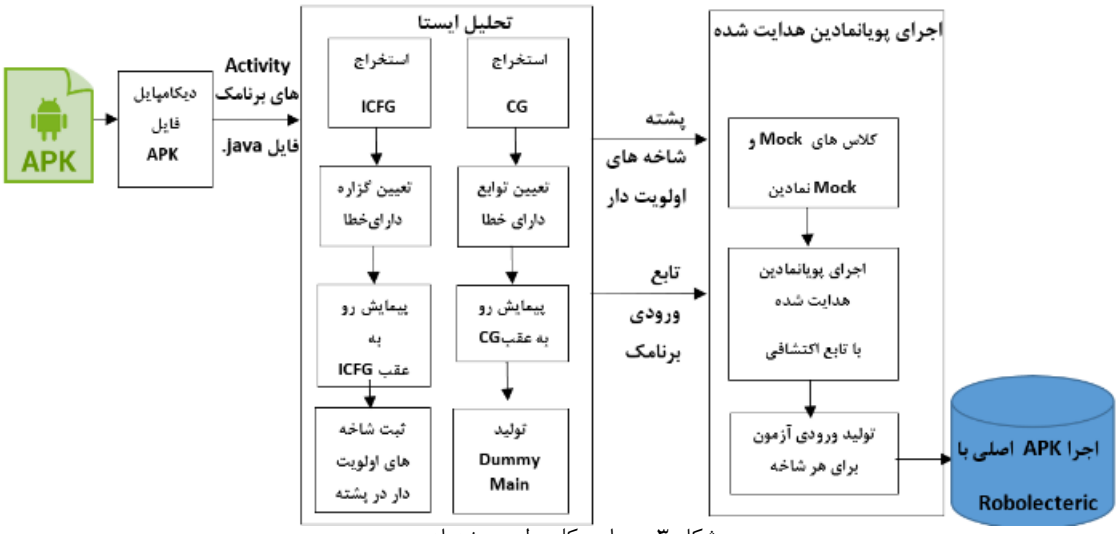
۲-۲- چالش‌های اجرای پویا-نمادین برای اندروید

برنامک‌های اندرویدی برروی ماشین مجازی Dalvik اجرا می‌شوند، درواقع کد منبع برنامک‌ها پس از کامپایل به جای بایت‌کد جاوا ، به بایت کد Dalvik تبدیل می‌شود. از آن جا که موتور اجرای پویا-نمادین برای بایت‌کد Dalvik وجود ندارد ما از موتور اجرای پویا-نمادین جاوا استفاده می‌کنیم ولی با دو چالش علمی مواجه می‌شویم. چالش اول آن است که برنامک‌های اندروید برخلاف برنامه‌های دیگر نقطه ورود به برنامه ندارند. این چالش را در بخش ۳-۲-۱ و با استفاده از گراف فراخوانی توابع حل کرده‌ایم.

چالش دوم آن است که برنامک‌های اندرویدی به چارچوب سیستم عامل اندروید و کتابخانه‌های SDK بسیار وابسته هستند. این واقعیت باعث ایجاد چالش واگرایی مسیر[۳] می‌شود. به طور کلی مشکل واگرایی مسیر زمانی رخ می‌دهد که متغیرهای نمادینی که برای اجرای پویا-نمادین تعریف کرده‌ایم از کد اصلی برنامک خارج شده و برای اجرا به کتابخانه‌های خارجی می‌روند. واگرایی مسیر در اجرای پویا-نمادین موجب می‌شود تا شاخه‌هایی از مسیر اجرا انتخاب گردد که در عمل کد کتابخانه خارجی آن وجود ندارد. ضمن آنکه در صورت وجود کد کتابخانه خارجی نیز آزمون کد کتابخانه هدف پژوهش ما نیست. درواقع فرض می‌شود که کتابخانه SDK و سیستم عامل اندروید نوشته شده کاملا سالم هستند و ما تنها نیاز به آزمون کد برنامک اندروید داریم. برای حل این چالش از ایده Mock استفاده کرده‌ایم و کلاس‌های SDK را به صورت Mock پیاده‌سازی کرده‌ایم.

چالش سوم آن است که، برنامک‌های اندرویدی رخدادمحور هستند، این جمله به این معناست که اغلب ورودی‌های یک

[8] Concrete
[9] Constraint Solver



شکل ۳: معماری کلی طرح پیشنهادی

برنامک از جنس رخداد هستند. درواقع هر مسیر اجرایی از این برنامک‌ها وابسته به رخدادهایی مبتنی بر تعامل کاربر یا رخدادهایی ناشی از دیگر برنامک‌ها است. لمس یک نقطه از نمایشگر گوشی، فشردن یک دکمه صفحه کلید و رسیدن یک پیامک نمونه‌هایی از رخداد در اندروید هستند. رخدادمحوربودن برنامک‌های اندرویدی موجب می‌شود تا موتور اجرای پویا-نمادین در هر اجرا از برنامه منتظر تعامل کاربر و یا رسیدن رخداد مورد نظر بماند تا پس از آن بتواند ادامه مسیر را اجرا کند. برای حل این مشکل از ایده ساخت Mock برای کلاس‌های تولید رخداد در SDK استفاده می‌کنیم.

۳- طرح پیشنهادی

معماری کلی طرح پیشنهادی ما را در شکل ۳ مشاهده می‌نمایید. ابزار ما از چهار بخش کلی تشکیل شده است. در بخش اول فایل apk برنامک را دیکامپایل می‌نماییم، و سپس در بخش دوم به تحلیل ایستا آن می‌پردازیم. خروجی تحلیل ایستا، تعیین نقطه ورودی برنامک و پشته حاوی شاخه‌های اولویت‌دار است. با استفاده از خروجی تحلیل ایستا، اجرای پویا-نمادین همراه با هیوریستیک ارائه شده اعمال می‌گردد، تا ورودی آزمون را به دست آوریم. در بخش چهارم، برنامک اصلی را با ورودی‌های عینی و ابزار Robolecteric می‌آزماییم. در ادامه بخش‌های مختلف معماری کلی شرح داده می‌شود.

۳-۱- دیکامپایل برنامک

در این بخش با استفاده از ابزار APKTool فایل apk برنامک را دیکامپایل می‌نماییم. خروجی این بخش درواقع کلاس‌های برنامک است که به صورت فایل .java تولید می‌گردد.

۳-۲- تحلیل ایستا

در بخش تحلیل ایستا ابتدا نقطه ورودی برنامک را استخراج می‌نماییم. سپس با پیمایش روبه‌عقب گراف کنترل جریان بین تابعی، پشته شاخه‌های اولویت‌دار را تعیین می‌کنیم. هر یک از این دو مورد در ادامه شرح داده خواهند شد.

۳-۲-۱-استخراج نقطه ورودی برنامه

برنامک‌های اندروید برخلاف برنامه‌های دیگر نقطه شروع مشخصی ندارند. یک برنامک اندروید می‌تواند چندین نقطه شروع داشته باشد که با توجه به رخدادهای متفاوت ایجاد شده، برنامک از یکی از آن نقطه‌ها آغاز می‌شود. در اجرای پویا-نمادین ما نیاز داریم تا از یک نقطه شروع مشخص کار را آغاز کنیم. به همین دلیل ابتدا گراف فراخوانی توابع برنامک را استخراج می‌کنیم. در این پژوهش ما مسئله یافتن خطا را به طور عام بررسی کرده‌ایم، ولی به عنوان نمونه برای نشان دادن صحت کارکرد هیوریستیک و تابع نقطه ورودی برنامک، «استثنای زمان‌اجرا[10]» را انتخاب کرده‌ایم. توجه گردد برای اینکه سایر خطاها مانند «خطای نشت حافظه» را نیز بتوانیم کشف کنیم صرفا کافی است تحلیل ایستای متناسب با آن به ابزار اضافه شود.

استثنای زمان‌اجرا می‌تواند از جنس «خطای تقسیم بر صفر»، «استثنای نقض محدوده آرایه» یا موارد دیگری باشد که در زمان اجرای برنامک اعلام[11] می‌شود. در برنامک‌های مورد آزمون، در نقاط مناسب برنامک، کد تولیدکننده این استثنا را قرار می‌دهیم.

[10] Runtime Ecxeption
[11] Throw

برای تولید تابع نقطه ورودی به برنامک باید گراف فراخوانی توابع را پیمایش نمود. اگر این گراف را به صورت روبه‌جلو و کامل پیمایش کنیم، می‌توانیم به حداکثر پوشش کد دست یابیم. اما با توجه به اینکه یافتن خطا مهمتر از پوشش حداکثری کد است، ما در اینجا ایده پیمایش روبه‌عقب گراف فراخوانی توابع، از تابع دارای خطا به ریشه را مطرح می‌کنیم. گراف فراخوانی توابع را با ابزار Soot [۹] بدست آورده‌ایم. گراف بدست‌آمده از ابزار شامل تمام حالت‌های ممکن برای اجرای برنامک است که هر حالت، مسیری از گره ریشه به برگ است. Soot گره ریشه را می‌سازد تا گراف به شکل همبند درآید. سپس الگورتیم پیمایش روبه‌عقب پیشنهادی خودمان را روی گراف استخراج شده اعمال کرده‌ایم. نمونه تابع نقطه شروع به برنامک در شکل ۴ دیده می‌شود. به ازای تمام مسیرهای مطلوب موجود در گراف، ما تابع نقطه شروع به برنامک را تولید می‌کنیم.

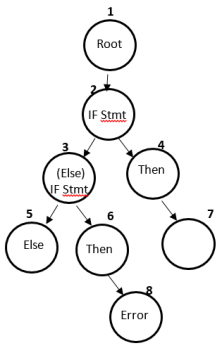
```
1:public class DummyMain {
2: public static void main(String[] args) {
3:   MunchLifeActivity mla=new MunchLifeActivity();
4:   mla.onCreate(null);
5:   mla.onStart();
6: }
7:}
```

شکل ۴: نمونه تابع نقطه شروع برنامک

۳-۲-۲-تعیین پشته شاخه‌های اولویت‌دار

برای بدست‌آوردن اولویت اجرای هر شاخه، گراف کنترل جریان بین تابعی برنامک را نیاز داریم. این گراف را با کمک ابزار Soot بدست می‌آوریم. این گراف در واقع از زیرگراف‌های کنترل جریان هر تابع و ارتباط بین آن‌ها تشکیل شده است. ما با ارائه الگوریتمی از عبارت رخداد خطا تا عبارت ریشه در گراف را به صورت روبه‌عقب پیمایش می‌کنیم و در این حین اطلاعات مربوط به انتخاب شاخه‌های مختلف در گراف را در یک پشته ذخیره می‌کنیم. در این نوشتار ما این پشته را «پشته شاخه‌های اولویت‌دار» می‌نامیم. این اطلاعات شامل دستور شرطی مورد نظر و اولویت شاخه‌های then و else نسبت به هم است. شاخه‌ای بر دیگری اولویت پیدا می‌کند که در پیمایش روبه‌عقب گفته شده، سریع‌تر پیمایش شود. سپس این پشته را به عنوان ورودی به اجرای پویا-نمادین خواهیم داد.

برای مثال در شکل ۵ نمونه این گراف آمده است. در این گراف از گره خطا (گره ۸) به صورت روبه‌عقب پیمایش به سمت گره ریشه (گره ۱) آغاز می‌کنیم. در گرهی که دستور شرطی وجود دارد، دستور شرطی همراه با اینکه شاخه then بر else اولویت دارد را در پشته ذخیره می‌کنیم. در این مثال در گره ۳، دستور شرطی و اولویت شاخه then بر else را به پشته اضافه می‌کنیم. همچنین برای گره ۲، دستور شرطی و اولویت else بر then را به پشته اضافه خواهیم کرد.



شکل ۵: مثالی از گراف کنترل جریان بین تابعی

۳-۳- تولید کلاس‌های Mock و Mock نمادین

برای اینکه برنامک روی JVM قابل اجرا باشد و چالش رخدادمحور بودن را حل کنیم، از کلاس‌های Mock به جای کلاس‌های اصلی SDK استفاده کرده‌ایم. چالش‌های گفته شده در بخش ۲-۲ مطرح شده‌اند. همچنین اگر جایی نیاز به رخدادی مانند فشردن دکمه توسط کاربر باشد، این رخداد را در تابع ورودی به برنامک با فراخوانی تابع Mock مرتبط با آن شبیه‌سازی می‌کنیم.

برای اینکه بتوانیم ورودی‌های آزمون را تولید کنیم، لازم است تا کلاس‌های از SDK که از کاربر داده دریافت می‌کنند،(مثل EditText) را به شکل Mock نمادین تولید کنیم. Mock نمادین کلاس Mock است که تمام متغیرها و تمام خروجی‌های تابع‌های آن به شکل نمادین هستند. این کار باعث می‌شود تا به درستی ورودی‌های که قرار است کاربر وارد کند، بعد از اجرای پویا-نمادین بدست بیایند.

۳-۴- اجرای پویا-نمادین هدایت شده با هیوریستیک

از مشکلات جدی که اجرای پویا-نمادین با آن روبه‌رو است مشکل انفجار مسیر می‌باشد. درواقع وقتی در درخت اجرای برنامه رو به پایین حرکت می‌کنیم، تعداد شاخه‌های اجرایی برنامه به طور نمایی زیاد می‌شود. از این رو اجرای پویا-نمادین برای برنامه‌های واقعی دچار مشکل کمبود زمان و منابع سیستم می‌گردد. در کارهای پیشین اجرای پویا-نمادین در اندروید نیز این چالش جدی وجود داشته ولی راهکار کارآمدی برای آن ارائه نشده است.

در این پژوهش ما یک هیوریستیک را معرفی می‌کنیم که از انفجار مسیر در اجرای پویا-نمادین برنامک‌های اندرویدی

جلوگیری می‌کند. در این هیوریستیک راهکار پیشنهادی ما بر دو ایده استوار است:

الف) اجرای پویا-نمادین برنامک را به تابع‌هایی نقطه شروعی که به خطا منتهی می‌شوند، محدود می‌کنیم.

ب) با استفاده از گراف کنترل جریان بین تابعی، اجرای پویا-نمادین برنامک را هدایت‌شده می‌نماییم.

ایده الف را در بخش ۳-۲-۱ و ایده ب را در بخش ۳-۲-۲ شرح داده‌ایم. برای اجرای پویا-نمادین از SPF استفاده کرده‌ایم. در SPF به صورت پیش‌فرض درخت اجرا و کد برنامه پیمایش عمق‌اول می‌شود و هیچ اولویت‌گذاری روی انتخاب شاخه‌های مختلف وجود ندارد. این موضوع ممکن است باعث شود که خطا در آخرین پیمایش و در آخرین شاخه اجرا شده در درخت اجرا کشف شود. در این پژوهش برای بهبود این موضوع، ما از تحلیل ایستا استفاده کرده‌ایم. ما از تحلیل ایستای خودمان نقطه شروع به برنامک (بخش ۳-۲-۱) و پشته شاخه‌های اولویت‌دار (بخش ۳-۲-۲) را به عنوان ورودی به بخش اجرای پویا-نمادین می‌دهیم.

برای این که اجرای پویا-نمادین متناسب با اولویت‌های انتخاب شاخه‌ها صورت پذیرد، الگوریتمی را ارائه داده‌ایم که به جای پیمایش عمق‌اول، در هر دستور شرطی نظیر حلقه‌ها و پرش‌های شرطی، با استفاده از پشته تصمیم می‌گیریم که اولویت اجرا را به کدام یک از شاخه‌های پیش‌رو بدهیم. استفاده از این ایده باعث می‌شود که ابتدا مسیر منتهی به خطا زودتر اجرا شود.

۳-۵- اجرای برنامک با ورودی‌های عینی

پس از اجرای پویا-نمادین هدایت‌شده برای اطمینان از درستی روش و کشف خطا، برنامک را با ورودی‌های عینی بدست آمده از آن اجرا می‌کنیم. در این بخش از کد منبع برنامک استفاده می‌کنیم تا خطا را با اجرای واقعی برنامک نیز کشف و مشاهده کنیم. برای این منظور از ابزار Robolectric استفاده کرده‌ایم.

۴- ارزیابی

ارزیابی راه‌کار ارائه شده را در دو مرحله انجام دادیم. ابتدا برای نشان دادن این که روش درست کار می‌کند ۱۰ برنامک را مطرح و پیاده‌سازی کردیم. این برنامک‌ها را به منظور راستی‌آزمایی تولید تابع نقطه شروع، درستی پشته شاخه‌های اولویت‌دار و همچنین درست بودن فرایند تولید کلاس‌های Mock و درست بودن اجرای پویا-نمادین و ارتباط این مولفه‌ها با هم پیاده‌سازی کردیم. در این برنامک‌ها برای نشان دادن وجود خطا، استثنای زمان‌اجرا را در آنها قرار دادیم. در این ۱۰ برنامک مرحله به مرحله و از ساده‌ترین حالت تا شکل‌های پیچده را پیاده‌سازی کردیم. همچنین حالت‌های مختلف جریان داده در برنامک (مثلا استفاده از Intent) را پیاده‌سازی کردیم. با استفاده از این برنامک‌ها مولفه جدید پیاده‌سازی شده در SPF را آزمودیم. این مولفه را برای اضافه کردن هیوریستیک به اجرای پویا-نمادین پیاده‌سازی کرده بودیم.

برای ارزیابی راه‌کار پیشنهادی، ما دو سوال پژوهشی را مطرح کرده‌ایم و به آن‌ها پاسخ داده‌ایم: ۱-آیا ابزار ما قابلیت تولید ورودی آزمون برای برنامک‌های واقعی اندروید را دارد؟ ۲- ابزار پیشنهادی ما نسبت به Sig-Droid که آخرین ابزار آزمون نظام‌مند برنامک‌های اندرویدی است، چه مزیت‌هایی دارد؟

برای پاسخ به سوالات مطرح شده، چهار برنامک دنیای واقعی جدول ۱ را آزمودیم که برنامک‌های مورد آزمون ابزار Sig-Droid نیز هستند. این برنامک‌ها از مخزن F-Droid [۱۰] انتخاب شده‌اند. در جدول ۱ اطلاعات این برنامک‌ها آمده است که میزان پیچیدگی آنها را نشان می‌دهد.

جدول ۱: مشخصات برنامک‌های واقعی مورد آزمون

ردیف	نام برنامک	تعداد خطوط برنامه	تعداد Activity	دسته‌بندی
۱	MunchLife	۶۳۱	۲	سرگرمی
۲	JustSit	۸۴۹	۴	ورزشی
۳	AnyCut	۱۰۹۵	۴	ابزار
۴	TippyTipper	۲۹۵۳	۶	ابزار

در جدول ۲ اطلاعات مربوط به تحلیل برنامک‌های جدول ۱ با ابزار Sig-Droid و کار خودمان را آورده‌ایم. ما با تحلیل ایستا و استفاده از گراف فراخوانی توابع، اجرا را محدود به تابع‌هایی می‌کنیم که در رسیدن به خطا نقش دارند. همچنین با گراف کنترل جریان بین تابعی و پشته شاخه‌های اولویت‌داری که بدست می‌آوریم، مسیرهایی از تابع‌های مطلوب را اجرا می‌کنیم که به خطا می‌رسند. وجود همزمان این دو تحلیل زمان اجرا را به شدت کاهش می‌دهد.

جدول ۲: مقایسه ابزار ما با Sig-Droid

ردیف	نام برنامک	Sig-Droid و کار ما بدون محدودیت یافتن خطا		کار ما با یافتن خطا	
		پوشش	زمان(ms)	پوشش	زمان(ms)
۱	MunchLife	٪۷۴	۱۸۶	٪۴۰	۲۰
۲	JustSit	٪۷۵	۱۳۷	٪۴۱	۱۴
۳	AnyCut	٪۷۹	۱۷۹	٪۳۷	۲۰
۴	TippyTipper	٪۷۸	۴۸۴	٪۴۳	۶۰

لازم به ذکر است که منظور از پوشش و زمان در ستون کار ما با یافتن خطا، میزان پوشش کد و زمان اجرا تا رسیدن به خطا است و اجرا تا زمان پوشش حداکثری ادامه پیدا می‌کند.

در جدول ۳، مقایسه ابزارهای مختلف با کار ما بر اساس معیارهای موجود در مقالات [۲][۳][۴] آمده است. لازم به ذکر است که مسئله انفجار مسیر در ابزارهای Monkey و Swifthand مطرح نمی‌شود چون این دو ابزار مسیرهای مختلف موجود در کد را بررسی و اجرا نمی‌کنند.

جدول ۳: مقایسه ابزارهای مختلف با کار ما

معیار مقایسه / ابزار	روش جست‌وجو	انواع رخداد	ترکیب تحلیل ایستا و پویا	عدم انفجار مسیر
Monkey	بی‌قاعده	متن،سیستم،GUI	×	-
Swifthand	مبتنی‌برمدل	متن،GUI	×	-
Sig-Droid	نمادین	متن،GUI	×	×
کار ما	پویا-نمادین	متن،سیستم،GUI	✓	✓

۵- جمع بندی

در این مقاله ما روشی بر اساس اجرای پویا-نمادین هدایت‌شده همراه با تحلیل ایستا در برنامک‌های اندرویدی را ارائه کرده‌ایم. ابتدا چالش‌های مربوط به آزمون چارچوب کاری اندروید، چالش‌های مربوط به اجرای پویا-نمادین و راه‌کارهای مقابله با آن‌ها را بررسی کرده‌ایم. سپس طرح پیشنهادی خود را به تفصیل عنوان کردیم. در نهایت از میان ۱۴ برنامک تحلیل شده که ۴ مورد از آن‌ها برنامک‌های مورد آزمون Sig-Droid بودند، توانستیم مزیت این روش از نظر کارایی(سرعت بیشتر) را نسبت به آخرین ابزار آزمون نظام‌مند برنامک‌های اندرویدی (Sig-Droid) نشان دهیم.

روش پیشنهادی ما، تا حد زیادی به‌ایده‌ی کلاس‌های Mock وابسته است. در این کار ما این کلاس‌ها را به صورت دستی برای برنامک‌های آزمون، تولید کرده‌ایم که فرایندی زمان‌بر است. در آینده قصد داریم با استفاده از ایده‌های استفاده-شده در آزمون نرم‌افزار و استفاده از ابزار Robolectric این موضوع را برای کلاس‌های SDK به صورت خودکار حل کنیم. راهکار دیگر آن است که با تولید موتور اجرای پویا-نمادین روی بایت‌کد Dalvik به جای بایت‌کد جاوا، تولید کلاس‌های Mock را برای SDK به کلی حذف نمود. در بعضی از برنامک‌ها از کد Native در کنار کد جاوا برای پیاده‌سازی استفاده می‌شود. در این کار ما کد Native را پشتیبانی نمی‌کنیم. با این حال مجموعه‌ای بزرگ از برنامک‌ها به زبان جاوا است و با ابزار پیاده‌سازی شده کنونی می‌توانیم تعداد زیادی از برنامک‌ها را تحلیل نماییم. درآینده قصد داریم کدهای Native را نیز پشتیبانی کنیم.

برای آنکه بتوانیم سایر خطاهای مرتبط با برنامک‌های اندرویدی مانند «خطای نشت حافظه» یا «استثناهای بررسی نشده» را تشخیص دهیم، لازم است تنها در تحلیل ایستا اطلاعات مورد نیاز در زمان اجرای پویای مرتبط با آن را تشخیص داده و در پشته شاخه‌های اولویت‌دار نگهداری کنیم. همچنین برای بعضی از خطاها لازم است تا کلاس‌های Mock نمادین مرتبط با آن را تولید نماییم.

مراجع

[1] "cafebazar." [Online]. Available: http://developers.cafebazaar.ir/fa/. [Accessed: 19-Dec-2017].

[2] "Android Monkey." [Online]. Available: https://developer.android.com/guide/developing/tools/monkey.html. [Accessed: 10-Oct-2017].

[3] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "SIG-Droid: Automated system input generation for Android applications," *2015 IEEE 26th International Symposium on Software Reliability Engineering, ISSRE 2015*, pp. 461–471, 2016.

[4] C. Wontae, N. George, and S. Koushik, "Guided gui testing of android apps with minimal restart and approximate learning," *Acm Sigplan Notices*, 2013, vol. 48, pp. 623--640.

[5] P. Godefroid, N. Klarlund, and K. Sen, "DART : Directed Automated Random Testing," *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 40, no. 6, pp. 213–223, 2005.

[6] "Robolectric." [Online]. Available: http://robolectric.org/. [Accessed: 01-Jan-2017].

[7] C. S. Pasareanu and N. Rungta, "Symbolic PathFinder: Symbolic Execution of Java Bytecode," *Proceedings of the IEEE/ACM international conference on Automated software engineering*, vol. 2, pp. 179–180, 2010.

[8] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, pp. 366--381, 2000.

[9] P. Arzt, Steven and Rasthofer, Siegfried and Fritz, Christian and Bodden, Eric and Bartel, Alexandre and Klein, Jacques and Le Traon, Yves and Octeau, Damien and McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14*, vol. 49, no. 6, pp. 259–269, 2014.

[10] "F-Droid." [Online]. Available: https://f-droid.org/. [Accessed: 15-Oct-2017].