اجرای پویا-نمادین

برای تشخیص آسیب‌پذیری تزریق SQL در برنامک‌های اندرویدی[1]

احسان عدالت[1]، بابک صادقیان[2]

[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
basadegh@aut.ac.ir

چکیده

از اجرای پویا-نمادین برای آزمون نرم‌افزارهای مختلف استفاده می‌شود. 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 روشی ارائه می‌شود که با اجرای پویا-نمادین همراه تحلیل آلایش[3] به دنبال تشخیص آسیب‌پذیری تزریق SQL در برنامک‌های اندرویدی هستیم. در این کار با تحلیل ایستا، گراف فراخوانی توابع و پیمایش برعکس از تابع آسیب‌پذیر[4] تا تابع منبع[5]، نقطه شروع برنامه[6] را تولید کردیم و فرایند تحلیل را محدود به تابع‌های مسیرهای مطلوب یافته‌شده کردیم. همچنین در این کار با ایده استفاده از کلاس‌های Mock مسئله رخدادمحوربودن و سربار بالای آزمون برنامک‌ها را حل کرده‌ایم. برای ارزیابی راه‌کار ارائه شده، ابتدا ۱۰ برنامک را خودمان پیاده‌سازی کردیم که ۴تای آنها آسیب‌پذیر بودند و توانستیم همه را تشخیص دهیم. همچنین از مخزن F-Droid استفاده کردیم که شامل برنامک‌های متن‌باز است. ۱۴۰ برنامک را به دلخواه از این مخزن انتخاب کردیم، که از این میان ۷ برنامک را که آسیب‌پذیر به تزریق SQL بودند را توانستیم تشخیص دهیم.

کلمات کلیدی

اجرای پویا-نمادین، تشخیص آسیب‌پذیری، آسیب‌پذیری SQL، برنامک‌های اندرویدی.

۱- مقدمه

میزان آگاهی برنامه نویسان از نحوه توسعه امن نرم‌افزار یکسان نیست. این مورد باعث می‌شود نرم‌افزارهای تولید شده با تهدیدهای امنیتی نیز در بازار قرار گیرد که موجب نشت اطلاعات حساس کاربران و یا نقض حریم خصوصی آنها می‌شود. از این رو نیاز به وجود راه‌کارهایی برای خودکار کردن فرایند تشخیص وجود آسیب‌پذیری در نرم‌افزارها احساس می‌شود. در سال‌های اخیر ارائه و توسعه ابزارهای همراه گسترش پیدا کرده است که حجم زیادی از این ابزارها مبتنی بر سیستم عامل اندروید هستند. این موضوع باعث شده است که پژوهش‌های زیادی در رابطه با برنامک‌های اندرویدی صورت بگیرد.

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 محسوب می‌شوند. تفاوت عمده آنها با سایر برنامه‌ها، در هم تنیدگی 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 تشخیص آسیب‌پذیری را با چالش روبه‌رو می‌کند.

برای تشخیص آسیب‌پذیری در نرم‌افزار نیاز است تا کد برنامه تحلیل شود. 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 [۱] ارائه شد. برای اولین بار ACTEVE [۲] از اجرای پویا-نمادین برای آزمون برنامک‌های اندرویدی استفاده کرده است. در این کار با تغییر دادن SDK سعی شده است تا رخداد لمس صفحه به شکل نمادین تولید شود. علاوه بر آن با تولید جایگشت‌های مختلف از رخدادها، رشته‌های مختلف از رخدادهای پشت سر هم تولید می‌شوند. اشکال این کار، بررسی یک رخداد منحصر به فرد است. علاوه بر آن تولید رشته‌های مختلف از رخدادها و اجرای آنها باعث می‌شود که فرایند آزمون با انفجار مسیر روبه‌رو شود. در [۳] از ابزار پیشین برای تشخیص وجود بدافزارها استفاده شده است که همان اشکالات ACTEVE را به ارث برده است. در [۴] از اجرای پویا-نمادین برای کشف نقض حریم خصوصی در برنامک‌های اندرویدی استفاده شده است. این ابزار به تحلیل‌گر انسانی کمک می کند تا فرایند کشف نشت اطلاعات حساس سریع‌تر اتفاق بیفتد. در [۵] ابزار Sig-Droid برای آزمون برنامک‌های اندرویدی ارائه شده است. این ابزار نزدیک‌ترین کار موجود به کار ما است. در این ابزار سعی شده است برنامک‌ها روی JVM[۷] کامپایل شوند تا بتوان به کمک موتورهای اجرای نمادینِ جاوا، برنامک را آزمود. این ابزار 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.

بر اساس آخرین دانسته ما تاکنون، ابزاری برای تشخیص آسیب‌پذیری در برنامک‌های اندرویدی با اجرای پویا-نمادین ارائه نشده است. در منبع [۶] عنوان شده است که آسیب‌پذیری تزریق می‌تواند موجب نشت اطلاعات حساس کاربر شود. آسیب‌پذیری تزریق در اندروید می‌تواند تزریق دستور به پوسته سیستم عامل، تزریق دستورات SQL به پایگاه‌داده SQLite، تزریق کد جاوا اسکریپت به WebView و یا تزریق Intent باشد. در این پژوهش روش و ابزاری ارائه کرده‌ایم که می‌تواند انواع آسیب‌پذیری تزریق را تشخیص دهد. برای نمونه و نشان دادن درستی کار، ما آسیب‌پذیری تزریق SQL را مورد توجه قرار دادیم. در این کار سعی کردیم برنامک روی JVM کامپایل شود تا بتوانیم از موتور SPF استفاده کنیم. برای این منظور ابتدا تابع ورودی به برنامه را تولید کردیم. با استفاده از تحلیل ایستا و استخراج گراف فراخوانی توابع، می‌توان این تابع را داشت. برای کاهش هزینه اجرای ابزار، گراف را از تابع هدف (مثلا query) تا تابع منبع (مثلا EditText) به صورت برعکس پیمایش کردیم و تابع ورودی به برنامه را بر این اساس تولید کردیم. 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، برنامه کامپایل شده و اجرای عادی داشته باشد بدون اینکه سربار کلاس‌های اصلی را داشته باشد. برای اینکه هزینه اجرای ابزار را باز هم کمتر کنیم، با توجه به مسیری که از گراف فراخوانی توابع بدست آوردیم، کلاس‌هایی از برنامک که نیاز به اجرا نداشتند را به صورت Mock تولید کردیم. برای تشخیص آسیب‌پذیری تزریق SQL، ما مولفه‌ای به SPF اضافه کردیم. در این مولفه تحلیل آلایش پویا را به اجرای پویا-نمادین اضافه کردیم. برای اینکه بتوانیم گردش داده آلایش‌شده در برنامه را به درستی انجام دهیم، کلاس‌های Mock نمادین را برای کلاس پایگاه‌داده SQLite و کلاس‌های منبع به برنامه (مثلا EditText) تولید کردیم. این کلاس‌ها شامل همان توابع و کلاس‌های اصلی هستند با این تفاوت که بدنه آنها حذف شده و خروجی توابع آنها نمادین خواهد بود. خروجی تحلیل SPF شامل دنباله پشته برنامه تا تابع آسیب‌پذیر، شناسه تابع منبع، تصفیه شدن یا نشدن داده ورودی توسط برنامه‌نویس و شناسه تابع نشت داده‌ها (مثلا TextView) است. در نهایت با استفاده از این خروجی‌ها و تابع ورودی بدست آمده از تحلیل ایستا با ابزار Robolectric [۷] قابلیت بهره‌جو[۸]یی برنامک به تزریق SQL را بررسی کردیم.

مطالب این مقاله بدین ترتیب سازمان‌دهی شده‌است که در بخش ۲ به صورت مختصر در مورد اجرای پویا-نمادین، چالش‌های مربوط به تحلیل برنامک‌های اندرویدی و آسیب‌پذیری تزریق SQL به برنامک‌های اندرویدی صحبت خواهیم کرد. در بخش ۳ روش و ابزار پیشنهادی را به تفصیل توضیح خواهیم داد. در بخش ۴ ارزیابی‌های صورت گرفته روی برنامک‌های محک بررسی خواهد شد و در نهایت در بخش ۵ به نتیجه‌گیری خواهیم پرداخت.

۲- پیش زمینه

در این بخش به طور مختصر و همراه با یک مثال اجرای پویا-نمادین را شرح خواهیم داد. پس از آن در مورد چالش‌های مربوط به تحلیل برنامه‌های سیستم عامل اندروید صحبت خواهیم کرد. سپس آسیب‌پذیری تزریق SQL به این گونه برنامه‌ها را کوتاه بررسی خواهیم کرد.

۲-۱- اجرای پویا-نمادین

برای توضیح روش اجرای پویا-نمادین از شکل ۱ که شامل یک برنامه ساده است استفاده خواهیم کرد. در روش پویا-نمادین متغیرها علاوه بر شکل نمادین به صورت مقدار عینی در نظر گرفته می‌شوند. مقدار عینی در ابتدا به صورت دلخواه انتخاب می‌شود. شرط مسیر، مجموعه‌ای از قیدهای استخراج شده از عبارت‌های شرطی موجود در برنامه است. برای مثال در این برنامه مقدار اولیه عینی x=1 و y=1 به صورت دلخواه انتخاب می‌شود. شرط مسیر استخراج شده با این ورودی‌ها pc=(y<5) خواهد بود. برای تولید ورودی عینی جدید شرط مسیر به حل‌کننده قید داده می‌شود. مقدار جدید y=6 و x=1 خواهد بود که با آن شرط مسیر pc=(y>5) and (x*x*x<10) تولید می‌شود. «حل‌کننده قید[۹]» توانایی حل عبارت‌های غیر خطی مثل (x*x*x<10) را ندارد. در این جا اجرای پویا-نمادین با قرار دادن یک مقدار دلخواه به جای x به اجرا ادامه می‌دهد. اگر مقدار دلخواه موجب درست شدن شرط مسیر شود خطا در برنامه کشف خواهد شد. مثلا x=3 و y=6 باعث می‌شود برنامه به خط ۴ برسد. به مجموعه شرط‌های مسیرِ استخراج شده، درخت اجرای برنامه گفته می‌شود. درخت اجرا ورودی‌های مناسب برای اجرای تمام مسیرهای برنامه را در اختیار ما قرار می‌دهد.

```
1:  testMe(int x, int y){
2:    if(y>5){
3:          if(x*x*x > 10){
4:                assert(false);
5:          }
6:    }
7:  }
8:  void main (){
9:     int x=symbolic_input();
10:   int y=symbolic_input();
11:   testMe(x,y);
12:}
```

شکل (۱) : نمونه برنامه ساده

با روش پویا–نمادین می‌توان علاوه بر پوشش مناسب مسیرهای مختلف از برنامه، برنامه را به منظور تحلیل، اجرا کرد. اجرای برنامه باعث می‌شود که مثبت نادرست وجود نداشته باشد و همچنین با روش‌های ضد ایستا مقابله کرد. چالش‌هایی که در این روش وجود دارد و راه‌حل‌های ما عبارتند از: الف) انفجار مسیر است، که در این مقاله با ایده تولید کلاس‌های Mock برای برخی از کلاس‌های برنامک با این مشکل مقابله کرده‌ایم. ب) چارچوب کاری و مدل‌سازی محیط است که در این پژوهش کلاس‌های SDK را به شکل Mock پیاده‌سازی کردیم تا رفتار برنامک اندرویدی و مسئله رخدادمحور بودن در آن حل شود. اشکالات ناشی از این چالش در بخش ۲–۲ توضیح داده شده‌اند.

برای اجرای نمادین برنامه‌های به زبان جاوا، ابزار SPF [۸] ارائه شده است. با استفاده از این ابزار می‌توان به صورت دلخواه مشخص کرد که چه تابع یا متغیری نمادین باشد. همچنین این ابزار از تعداد زیادی از حل کننده‌های قید پشتیبانی می‌کند که با استفاده از آنها می‌توان قیدهای مختلف را تحلیل کرد. به طور خاص برای رشته‌ها که در تحلیل ما بسیار اهمیت دارد، چند حل‌کننده قید با قدرت‌های مختلف در SPF وجود دارد. علاوه بر آن این ابزار اجرای پویا–نمادین را نیز پشتیبانی می‌کند و این موضوع باعث کشف تعداد بیشتری از خطاها در برنامه می‌شود. در این مقاله با تغییر SPF به دنبال تشخیص آسیب‌پذیری تزریق SQL به برنامک‌های اندرویدی هستیم.

۲–۲- چالش‌های مربوط به تحلیل برنامک‌های اندرویدی

برنامک‌های اندروید وابسته به مجموعه‌ای از کتابخانه‌هایی هستند که در بیرون از دستگاه یا شبیه‌ساز در دسترس نیستند. کد اندروید در DVM[۱۰] اجرا می‌شود. برخلاف کدهای برنامه‌های جاوا که در JVM اجرا می‌شوند. پس به جای بایت‌کد جاوا برنامک‌های اندروید به بایت‌کد Dalvik کامپایل می‌شوند. همچنین برخلاف برنامه‌های جاوا که یک نقطه شروع به برنامه دارند، برنامک‌ها نقطه شروع مشخصی ندارند. برای اجرای کدهای اندروید در JVM لازم است تا نقطه شروع فرضی(dummyMain) برای آن تولید شود. برای این منظور از تحلیل ایستا و گراف فراخوانی توابع استفاده کرده‌ایم.

برنامک‌های اندروید بسیار وابسته به کتابخانه‌های SDK هستند و این موضوع باعث ایجاد مشکل واگرایی مسیر می‌شود[۵]. در اجرای پویا–نمادین اگر یک مقدار نمادین از محدوده برنامک در حال تحلیل خارج شود، مثلا برای انجام یک پردازش در اختیار SDK قرار گیرد، گفته می‌شود که واگرایی مسیر اتفاق افتاده است. از آنجایی که فرض می‌شود SDK به صورت امن پیاده‌سازی شده است، اجرای پویا–نمادین آن تنها موجب ایجاد سربار اضافی در فرایند تشخیص آسیب‌پذیری خواهد شد. برای حل این موضوع کلاس‌های SDK را به شکل Mock استفاده کرده‌ایم.

برنامک‌های اندروید رخدادمحور هستند. به این معنی که در اجرای پویا–نمادین، مولفه اجرا باید منتظر کاربر بماند تا با تعامل با برنامک یک رخداد مثل لمس صفحه نمایش ایجاد شود. پس نمی‌توان در محیط اندروید این گونه تحلیل‌ها را انجام داد و باید رفتار محیط اجرای برنامک و کاربر را به گونه‌ای شبیه‌سازی کرد. با استفاده از کلاس‌های Mock به جای کلاس‌های اصلی این موضوع را حل کرده‌ایم.

۳–۲- آسیب‌پذیری تزریق SQL به برنامک‌های اندرویدی

برنامک‌های اندرویدی می‌توانند از پایگاه‌داده داخلی SQLite استفاده کنند. SDK کتابخانه‌ای به همین منظور ارائه داده است که امکاناتی برای پیاده‌سازی امن و همچنین ناامن در آن وجود دارد. در شکل ۲ نمونه برنامه‌ای با دو شکل استفاده امن (خط ۳) و ناامن (خط ۲) از این تابع‌ها آورده شده است. ابتدا ورودی از کاربر توسط editText گرفته می‌شود و به تابع rawQuery برای پرس‌وجو داده می‌شود. سپس خروجی در textView نمایش داده می‌شود. در [۹] راه‌های وقوع و مقابله با این آسیب‌پذیری به صورت کلی آمده است. از جمله راه‌کارهای مقابله، استفاده از تابع‌های پارامتری است که در SDK این امکان وجود دارد. هرگاه برنامه‌نویس از شکل پارامتری استفاده می‌کند، به جای مقادیری که قرار است کاربر وارد کند از نویسه ’?‘ استفاده می‌کند و ورودی‌های کاربر را به عنوان ورودی تابع آسیب‌پذیر قرار می‌دهد. (خط ۳) با این کار ورودی کاربر به طور مستقیم با دستور برنامه‌نویس جمع نمی‌شود.

```
1: String st  =  editText.getText().toString();
2: Cursor c = db.rawQuery("SELECT * FROM student where
      stdno = '"+st+"'", null);          //not Secured way
3: Cursor c = db.rawQuery("SELECT * FROM student where
      stdno=?", new String[]{st});       //Secured way
…
4: textView.setText(buffer);
```

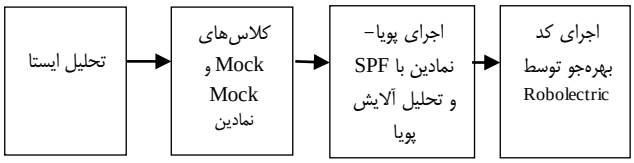
شکل (۲) : نمونه کد آسیب‌پذیر و امن اندروید به تزریق SQL

برای تشخیص آسیب‌پذیری تزریق SQL سه مورد باید بررسی شود: ۱) از ورودی برنامه به تابع‌های آسیب‌پذیر مسیری وجود داشته باشد. ۲) از شکل پارامتری این تابع‌ها استفاده نشده باشد. ۳) خروجی تابع آسیب‌پذیر به تابع‌های خاص که موجب نشت می‌شوند، داده شود.

در مرجع [۱۰]، نویسنده حمله تزریق دستور به پوسته سیستم عامل اندروید و تزریق دستور SQL به SQLite را بررسی کرده است. در آن مقاله، برای مقابله با این آسیب‌پذیری‌ها روش دنبال‌کردن آلایش استفاده شده است. برای پیاده‌سازی، گفته شده که باید سیستم عامل اندروید تغییر پیدا کند تا بتوان رشته‌های آلایش‌یافته را دنبال کرد. ما در این مقاله بدون تغییر سیستم عامل به این هدف خواهیم رسید.

۳- روش و ابزار پیشنهادی

در شکل ۳ نمای کلی از مراحل پیشنهادی برای تشخیص آسیب‌پذیری تزریق در برنامک‌های اندروید آورده شده است. برای انجام کار از مجموعه‌ای از ابزارها، علاوه بر الگوریتم‌های طراحی شده خودمان استفاده کرده‌ایم. در ادامه این فرایند به طور کامل توضیح داده خواهد شد.



شکل (۳) : مراحل اجرا در ابزار

در این پژوهش برای تشخیص آسیب‌پذیری تزریق از تحلیل آلایش همراه با اجرای پویا-نمادین استفاده کرده‌ایم. در تحلیل ما، هرجا که داده‌ای نمادین باشد، نشان دهنده این موضوع است که آن داده آلایش شده است. این موضوع باعث می‌شود تحلیل آلایش با اجرای پویا-نمادین ترکیب شوند. برای اینکه تحلیل آلایش صورت پذیرد لازم است تا ورودی‌های به برنامه نمادین شوند. برای این منظور ما ایده استفاده از کلاس Mock نمادین را پیشنهاد می‌کنیم. دلیل استفاده از ایده Mock نمادین این است که تابع‌های ورودی بخشی از SDK هستند و برنامه‌نویس صرفا از آنها استفاده می‌کند. برای اینکه برنامک اجرا شود لازم است که این کلاس‌ها Mock شوند و برای اینکه تحلیل کامل شود باید مقادیر متغیرها در آن و خروجی تابع‌های آن نمادین باشند. سپس برنامک را به شکل پویا-نمادین اجرا می‌کنیم. در حین اجرا، وضعیت نمادین‌بودن متغیرها را ذخیره می‌کنیم. برای مثال در str = str1+str2; (جمع دو رشته) که str1 نمادین و str2 عینی است، بعد از اجرای دستور، str را نمادین در نظر می‌گیریم. هرگاه اجرا به تابع آسیب‌پذیر برسد، با توجه به اطلاعات ذخیره‌شده در رابطه با متغیرها، نمادین بودن ورودی تابع آسیب‌پذیر را بررسی می‌کنیم. همچنین در حین اجرا تصفیه‌شدن ورودی توسط برنامه‌نویس را بررسی می‌کنیم. مثلا اگر str=str2; شود str عینی می‌شود و ما آن را نمادین در نظر نخواهیم گرفت. به زبان ساده برنامه‌نویس خود روندی برای تصفیه تولید کرده است. این کار باعث می‌شود مقدار نمادین به تابع آسیب‌پذیر نرسد. علاوه بر آن استفاده شدن از تابع پارامتری توسط برنامه‌نویس به عنوان روشی برای تصفیه کردن داده‌ها بررسی می‌شود.

برای اینکه زنجیره آسیب‌پذیری تزریق کامل شود، لازم است تا خروجی تابع آسیب‌پذیر به تابع نشت وارد شود. ادامه تحلیل آلایش تا تابع نشت برای برخی از آسیب‌پذیری‌های تزریق فرایندی سخت‌گیرانه است ولی برای اینکه تحلیل کامل شود و ما بتوانیم تمام آسیب‌پذیری‌های تزریق را پوشش دهیم، ما آن را انجام داده‌ایم. برای اینکه تحلیل آلایش را بتوان ادامه داد، Mock نمادین را برای کلاس آسیب‌پذیر نیز بایستی تولید کرد. کلاس‌های آسیب‌پذیر هم کلاس‌هایی از SDK هستند و باید به شکل Mock نمادین آنها را تولید کنیم. اجرای پویا-نمادین ادامه پیدا می‌کند تا زمانی که به تابع نشت برسیم، اگر منشا ورودی به این تابع از کلاس Mock نمادین آسیب‌پذیر باشد، پس از تابع منبع به تابع آسیب‌پذیر و سپس به تابع نشت مسیر وجود داشته است. وجود این مسیر یعنی اینکه آسیب‌پذیری تزریق در برنامکِ مورد تحلیل وجود دارد.

۳-۱- تحلیل ایستا

همان طور که گفته شد، برنامک‌های اندرویدی مثل برنامه‌های مرسوم به زبان جاوا دارای نقطه شروع به برنامه نیست. برای اینکه بتوانیم از SPF استفاده کنیم لازم است تا نقطه شروع به برنامه را تولید کنیم. برای این کار از گراف فراخوانی توابع استفاده کرده‌ایم. برای بدست آوردن این گراف از ابزار soot [۱۱] استفاده می‌کنیم. علاوه بر آن برای اینکه از مسئله انفجار مسیر در اجرای پویا-نمادین جلوگیری کنیم، گراف فراخوانی توابع را برای یافتن تابع‌های آسیب‌پذیر (مثلا query) به صورت عمق-اول جست‌وجو می‌کنیم. هنگامی که تابع آسیب‌پذیر پیدا شد، گراف را به صورت برعکس پیمایش می‌کنیم تاجایی که به ورودی‌های به برنامه (مثلا EditText) برسد. سپس برای هر مسیر یافته‌شده توالی تابع‌ها برای فراخوانی و نوع آنها را مشخص می‌کنیم. نوع تابع می‌تواند «Normal» یا «Listener» باشد. تابع‌های از نوع Listener موجب ایجاد رخداد در برنامه می‌شوند. (مثلا تابع onCreate از نوع Normal و تابع onClick از نوع Listener است.) اگر تابع از نوع Listener باشد، داده لازم برای فراخوانی شی مرتبط با آن (مثلا شناسه دکمه روی صفحه یعنی R.id.button) را نیز استخراج می‌کنیم. با مجموعه این داده‌ها کلاس dummyMain تولید می‌شود.(شکل ۴ الف)

۳-۲- تولید کلاس‌های Mock و Mock نمادین

در این کار ما به دو منظور از کلاس‌های Mock استفاده کرده‌ایم. برای اینکه برنامه روی JVM اجرا کنیم و مسئله رخدادمحور بودن را حل کنیم، از کلاس‌های Mock به جای کلاس‌های SDK استفاده کردیم. همچنین اگر جایی نیاز به رخدادی توسط کاربر باشد (مثل فشردن دکمه) آن رخداد را در dummyMain با فراخوانی تابع Mock مرتبط با آن (مثلا performClick) شبیه‌سازی می‌کنیم. همچنین برای حل مسئله انفجار مسیر، تنها کلاس‌هایی که از تحلیل ایستا استخراج کردیم را تحلیل می‌کنیم و بقیه کلاس‌های برنامک را Mock می‌کنیم. در هر دوی این حالت‌ها وجود کلاس Mock هزینه و سربار تحلیل را کاهش می‌دهد.

همان‌طور که گفته شد، برای اینکه تحلیل آلایش ما به شکل کامل صورت پذیرد، باید کلاس مربوط به تابع آسیب‌پذیر و تابع منبع را به شکل Mock نمادین تولید کنیم. در این پژوهش، ما برای اولین بار ایده کلاس Mock نمادین را مطرح می‌کنیم بدین ترتیب که کلاس Mock نمادین کلاسی است که تابع‌های کلاس اصلی را دارد با این تفاوت که بدنه تابع حذف می‌شود و خروجی تابع‌ها نمادین خواهند بود. این ایده باعث می‌شود تحلیل آلایش و گردش داده‌های آلایش‌شده در برنامه صورت بگیرد و وابستگی به چارچوب کاری اندروید و یا برنامه‌های دیگر حذف شود. چون کلاس‌های منبع و آسیب‌پذیر هر دو بخشی از SDK هستند، پس از این جهت به شکل Mock پیاده‌سازی شوند. همچنین چون مقادیر متغیرها و خروجی تابع‌های آنها در تحلیل آلایش استفاده می‌شوند، (یعنی داده‌هایی آلایش‌شده هستند) پس باید این مقادیر و خروجی‌ها را نمادین کنیم.

```
1: public class dummyMain {
2:   public static void main(String[] args) {
3:     MainActivity ma=new MainActivity();
4:     ma.onCreate(null);
5:     Button b= (Button) ma.findViewById(R.id.button);
6:     b.performClick();
7:   }
8: }
```

الف

```
1: public void SqlInjection_Exploitability() throws Exception {
2:    Activity ma = Robolectric.setupActivity(MainActivity.class);
3:    Button b= (Button) ma.findViewById(R.id.button);
4:    EditText et = (EditText) ma.findViewById(R.id.editText);
5:    TextView tv = (TextView) ma.findViewById(R.id.textview);
6:    et.setText("a' or '1'='1");
7:    b.performClick();
8:    Logger.error((String) tv.getText(),null);
9:}
```

شکل (۴) : الف) نمونه dummyMain تولید شده برای اجرا درSPF . ب) نمونه کد بهره جو درRobolectric .

۳-۳-اجرای پویا-نمادین همراه با تحلیل آلایش توسط SPF اصلاح شده

برای تشخیص آسیب‌پذیری ورودی‌های برنامک (مثلا EditText) را نمادین در نظر گرفتیم. همچنین یک مولفه جدید به SPF اضافه کردیم. در این مولفه اجرای برنامه را ادامه می‌دهیم تا زمانی که به تابع آسیب‌پذیر (مثلا query) برسیم. سپس نمادین بودن ورودی تابع آسیب‌پذیری را بررسی می‌کنیم. در صورت نمادین بودن، پارامتری نبودن تابع آسیب‌پذیر را تشخیص می‌دهیم. در صورت برقرار بودن تمام این شرایط، خروجی تابع آسیب‌پذیر را نمادین می‌کنیم (Mock نمادین). سپس اجرا را ادامه می‌دهیم تا زمانی که به یکی از توابع نشت اطلاعات (مثلا TextView با هر رسانه‌ای که به بیرون داده میدهد مثل شبکه) برسیم. در صورتی که ورودی تابع نشت، از Mock نمادین آسیب‌پذیر باشد، اطلاعات مورد نیاز برای تحلیل را اعلام می‌کنیم. این اطلاعات شامل شناسه تابع منبع، شناسه تابع نشت، دنباله پشته برنامه تا تابع آسیب‌پذیر و تصفیه شدن یا نشدن داده ورودی توسط برنامه‌نویس است.

۴-۳-آزمون نرم‌افزار برای بررسی میزان بهره‌جویی

با اطلاعاتی که از تحلیل ایستا و پویا-نمادین بدست آوردیم، برنامه را با ابزار Robolectric می‌آزماییم. ابزار Robolectric به منظور آزمون برنامک‌های اندرویدی ارائه شده است که نیاز به اجرای برنامک در محیط اندروید ندارد. برای استفاده از این ابزار باید مسیر موردِ آزمون به آن داده شود. در شکل ۴ نمونه آن آمده است. برای این مورد ما از خروجی تحلیل ایستا استفاده کردیم و عملا کلاس dummyMain بدست آمده را به Robolectric دادیم (خط ۳ شکل الف با ۲ ب، خط ۵ الف با ۳ ب و ۶ الف با ۷ ب یکی است). همچنین لازم است تا رشته‌هایی مثل «a’ or ‘1’=’1» را به ورودی آسیب‌پذیر برنامه بدهیم تا بهره‌جویی از آن را بیازماییم. (خط ۶ شکل ب) برای بررسی بهره‌جویی، خروجی تابع نشت یعنی TextView را مشاهده کردیم (خط ۸ شکل ب). برای یافتن شناسه تابع ورودی آسیب‌پذیر و شناسه تابع نشت از خروجی تحلیل پویا-نمادین استفاده کردیم.

۴- ارزیابی روش با آزمون برنامک‌ها

ارزیابی راه‌کار ارائه شده را در دو مرحله انجام دادیم. ابتدا برای نشان دادن این که روش درست کار می کند ۱۰ برنامک را خودمان پیاده‌سازی کردیم[۱۱]. ۶ برنامک را به منظور راستی‌آزمایی تولید کلاس dummyMain، 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[۱۲] را در برنامک‌ها قرار دادیم. در این ۶ برنامک مرحله به مرحله و از ساده‌ترین حالت تا شکل‌های پیچیده را پیاده‌سازی کردیم. در ۴ برنامک باقی‌مانده آسیب‌پذیری تزریق SQL را به جای استثنای زمان‌اجرا قرار دادیم. در این برنامک‌ها سعی کردیم حالت‌های مختلف استفاده از تابع‌های آسیب‌پذیر را در دو حالت استفاده از حالت پارامتری و غیرپارامتری آنها پیاده‌سازی کنیم. همچنین حالت‌های مختلف جریان داده در برنامک (مثلا استفاده از Intent و یا استفاده از پایگاه‌داده برنامک دیگر) را پیاده‌سازی کردیم. با استفاده از این برنامک‌ها مولفه جدید پیاده‌سازی شده در SPF را آزمودیم. این مولفه را برای تشخیص آسیب‌پذیری تزریق SQL پیاده‌سازی کرده بودیم.

برای اینکه نشان دهیم روش ما در برابر برنامک‌های واقعی هم درست کار می‌کند، از مخزن F-Droid[۱۱] [۱۲] استفاده کردیم. این مخزن شامل برنامک‌های اندرویدی متن‌باز در موضوعات مختلف است. ۱۴۰ برنامک مختلف را به دلخواه انتخاب کردیم. برای تحلیل ابتدا سعی کردیم برای برنامک‌ها dummyMain بسازیم. از این برنامک‌ها، برای ۷ برنامک، dummyMain تولید شد. این نشان می‌دهد که در بقیه برنامه‌ها مسیری از تابع منبع به تابع آسیب‌پذیر یافت نشده است و این یعنی امکان آسیب‌پذیری در آنها وجود ندارد. ۷ برنامک گفته شده آسیب‌پذیر به تزریق SQL بودند که تنها در یک مورد برنامه‌نویس از تابع پارامتری استفاده کرده بود و جلوی وقوع آسیب‌پذیری را گرفته بود. برای اینکه از درستی نتایج مطمئن شویم ۷ برنامک بدست آمده را به صورت دستی هم تحلیل کردیم که نتایج بدست آمده با نتایج خروجی ابزار مطابقت داشت.

ابزار کار ما رخدادمحور بودن اندروید را پشتیبانی می‌کند. همچنین با ترکیب تحلیل ایستا و پویا توانستیم با انفجار مسیر در حین تحلیل، مقابله کنیم. علاوه بر این، ابزار ما می‌تواند آسیب‌پذیری تزریق را در برنامک‌های

اندرویدی تشخیص دهد. در جدول ۱ مقایسه ابزار ما با ابزارهای مشابه فعلی از ۵ جنبه مطرح در مقالات آمده است.[۲،۳،۵]

جدول(۱) : مقایسه ابزار ارائه شده با ابزارهای مشابه موجود

	رخدادمحور بودن	عدم انفجار مسیر	تشخیص بمب منطقی	تشخیص آسیب‌پذیری	ترکیب تحلیل ایستا و پویا
ACTEVE	✓	×	×	×	×
Condroid	✓	×	✓	×	×
Sig-Droid	✓	×	×	×	×
کار ما	✓	✓	×	✓	✓

۵- نتیجه‌گیری

در این مقاله ما روشی بر اساس اجرای پویا–نمادین همراه با تحلیل آلایش برای تشخیص آسیب‌پذیری تزریق SQL در برنامک‌های اندرویدی ارائه کردیم. در این مقاله ابتدا چالش‌های مربوط به چارچوب کاری اندروید، چالش‌های مربوط به اجرای پویا–نمادین و راه‌کارهای مقابله با آنها و ویژگی‌های مربوط به آسیب‌پذیری تزریق SQL را بررسی کردیم. سپس طرح پیشنهادی را به تفصیل عنوان کردیم و در نهایت از میان ۱۵۰ برنامک تحلیل شده ۱۱ برنامک آسیب‌پذیر به تزریق SQL یافت شدند.

روش ما بسیار به کلاس‌های Mock متکی است. در این کار ما این کلاس‌ها را دستی تولید کردیم که فرایندی زمان‌بر است. در آینده قصد داریم با استفاده از ایده‌های ابزار Robolectric این موضوع را برای کلاس‌های SDK به صورت خودکار حل کنیم. همچنین می‌توان با تولید کردن موتور اجرای پویا–نمادین روی بایت‌کد Dalvik به جای بایت‌کد جاوا بخش تولید کردن کلاس‌های Mock را برای SDK به کلی حذف کرد. در بعضی از برنامک‌ها از کد Native برای پیاده‌سازی استفاده می‌شود. در این کار تاکید ما بر کد جاوا بود و کد Native را پشتیبانی نمی کنیم. با این حال مجموعه‌ای بزرگ از برنامک‌ها به زبان جاوا است و با ابزار پیاده‌سازی شده کنونی می‌توانیم تعداد زیادی از برنامک‌ها را تحلیل کنیم. در این پژوهش از گراف فراخوانی توابع در تحلیل ایستای برنامک استفاده کردیم که باز موجب اجرای برخی از مسیرهای نامطلوب در برنامه می‌شود. در آینده می خواهیم از مجموع گراف فراخوانی توابع و گراف کنترل جریان در هر تابع استفاده کنیم. این موضوع باعث می‌شود، مسیر دقیق از تابع منبع به تابع آسیب‌پذیر به موتور پویا–نمادین داده شود و مسلما کارایی تحلیل افزایش پیدا خواهد کرد.

در آینده برای اینکه سایر آسیب‌پذیری‌های تزریق را هم تشخیص دهیم کافی است، تابع‌های آسیب‌پذیر، تابع‌های ورودی و تابع‌های نشت را مشخص کنیم و کلاس‌های Mock نمادین مرتبط با آن آسیب‌پذیری را نیز تولید کنیم و به ابزار بدهیم.

منابع

[1] Godefroid, P., Klarlund, N., Sen, K., "*DART : Directed Automated Random Testing*", PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Vol. 22, pp. 213–223, 2005.

[2] Anand, S., Naik, M., Harrold, M.J., Yang, H., "*Automated concolic testing of smartphone apps*", Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, p. 59, 2012.

[3] Schütte, J., Fedler, R., Titze, D., "*ConDroid : Targeted Dynamic Analysis of Android Applications*", Advanced Information Networking and Applications (AINA), 2015 IEEE 29th International Conference on, pp. 571–578, 2015.

[4] Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., Wang, X.S., "*AppIntent : Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection*", Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13, pp. 1043–1054, 2013.

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

[6] OWASP, *Mobile Security Testing Guide*, https://www.owasp.org/index.php/OWASP_Mobile_Security_Testing_Guide, October 2017.

[7] Robolectric, *Test-Drive Your Android Code*, http://robolectric.org/, September 2017.

[8] Pasareanu, C.S., Rungta, N., "*Symbolic PathFinder: Symbolic Execution of Java Bytecode*", Proceedings of the IEEE/ACM international conference on Automated software engineering- ASE '10, pp. 179–180, 2010.

[9] OWASP, *SQL Injection*, https://www.owasp.org/index.php/SQL_Injection, October 2017.

[10] Smith, G.J., *Analysis and Prevention of Code-Injection Attacks on Android OS*, M.Sc. Thesis, University of South Florida, 2014.

[11] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P., "*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, pp. 259–269, 2014.

[12] F-Droid, *Free and Open Source Software Applications for Android*, https://f-droid.org/, September 2017.

پانویس ها

[1] Android Apps
[2] Software Development Kit
[3] Taint Analysis
[4] Sink Method
[5] Source Method
[6] Main Method
[7] Java Virtual Machine
[8] Exploit
[9] Constraint Solver
[10] Dalvik Virtual Machine
[11] github.com/ehsan-is-everything/CIG-Droid/tree/apk-src
[12] Runtime Exception