دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر و فناوری اطلاعات

پایان‌نامه کارشناسی‌ارشد

گرایش امنیت اطلاعات

اجرای پویا-نمادین برای تشخیص آسیب‌پذیری تزریق به برنامه‌های کاربردیِ گوشی‌های هوشمند

نگارش

احسان عدالت

استاد راهنما

جناب آقای دکتر بابک صادقیان

آذر ۹۶

صفحه فرم ارزیابی و تصویب پایان نامه- فرم تأیید اعضاء کمیته دفاع

در این صفحه فرم دفاع یا تایید و تصویب پایان نامه موسوم به فرم کمیته دفاع- موجود در پرونده آموزشی- را قرار دهید.

نکات مهم:

- ✓ نگارش پایان نامه/رساله باید به زبان فارسی و بر اساس آخرین نسخه دستورالعمل و راهنمای تدوین پایان نامه های دانشگاه صنعتی امیرکبیر باشد.(دستورالعمل و راهنمای حاضر)
- ✓ رنگ جلد پایان نامه چاپی کارشناسی، کارشناسی ارشد و رساله دکترا باید به ترتیب مشکی، طوسی و سفید رنگ باشد.
- ✓ چاپ و صحافی پایان نامه/رساله بصورت پشت و رو(دورو) بلامانع است و انجام آن توصیه می شود.

به نام خدا

تعهدنامه اصالت اثر

تاریخ:

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

اینجانب احسان عدالت متعهد می‌شوم که مطالب مندرج در این پایان نامه حاصل کار پژوهشی اینجانب تحت نظارت و راهنمایی اساتید دانشگاه صنعتی امیرکبیر بوده و به دستاوردهای دیگران که در این پژوهش از آنها استفاده شده است مطابق مقررات و روال متعارف ارجاع و در فهرست منابع و مآخذ ذکر گردیده است. این پایان نامه قبلاً برای احراز هیچ مدرک هم‌سطح یا بالاتر ارائه نگردیده است.

در صورت اثبات تخلف در هر زمان، مدرک تحصیلی صادر شده توسط دانشگاه از درجه اعتبار ساقط بوده و دانشگاه حق پیگیری قانونی خواهد داشت.

کلیه نتایج و حقوق حاصل از این پایان نامه متعلق به دانشگاه صنعتی امیرکبیر می‌باشد. هرگونه استفاده از نتایج علمی و عملی، واگذاری اطلاعات به دیگران یا چاپ و تکثیر، نسخه‌برداری، ترجمه و اقتباس از این پایان نامه بدون موافقت کتبی دانشگاه صنعتی امیرکبیر ممنوع است. نقل مطالب با ذکر مآخذ بلامانع است.

احسان عدالت

امضا

در صورت تمایل این صفحات نیز اضافه شود: (اختیاری)

- **صفحه تقدیم**

نویسنده پایان‌نامه، درصورت تمایل می‌تواند برای سپاسگزاری پایان‌نامه خود را به شخص یا اشخاص و یا ارگان خاصی تقدیم نماید.

- **صفحه تقدیر و تشکر**

نویسنده پایان‌نامه می‌تواند مراتب امتنان خود را نسبت به استادراهنما و استادمشاور و یا دیگر افرادی که طی انجام پایان‌نامه به نحوی او را یاری و یا با او همکاری نموده‌اند ابراز دارد.

چکیده

واژه‌های کلیدی:

فهرست مطالب

صفحه

فهرست اشکال

صفحه

فهرست جداول

صفحه

فصل اول

مقدمه

مقدمه

فصل دوم

اجرای پویا-نمادین

اجرای پویا-نمادین

یکی از روش‌های آزمون نرم‌افزار اجرای پویا-نمادین است. در این روش به صورت هم‌زمان کد برنامه را هم به صورت عینی و هم به صورت نمادین اجرا می‌کنند. اجرای نمادین باعث می‌شود پوشش مناسبی از کد بدست بیاید ولی در عین حال ممکن است مسیری از برنامه با اجرای نمادین صرف، قابل دسترس نباشد که وجود اجرای عینی این مسئله را حل می‌کند. در این فصل قصد داریم اجرای پویا-نمادین و کارهای صورت گرفته در این حوزه را مورد بررسی قرار دهیم. در انتهای فصل کاربرد این روش در برنامک‌های اندرویدی نیز مورد بررسی قرار خواهد گرفت.

۱-۲- بیان اجرای نمادین و پویا-نمادین با مثال

برای توضیح روش اجرای نمادین و پویا-نمادین از شکل ۲-۱ که شامل یک برنامه ساده است استفاده خواهیم کرد. تفاوت دو اجرای نمادین و پویا-نمادین در این است که در اجرای پویا-نمادین علاوه بر اجرای نمادین به صورت هم‌زمان برنامه به شکل عینی نیز اجرا خواهد شد.

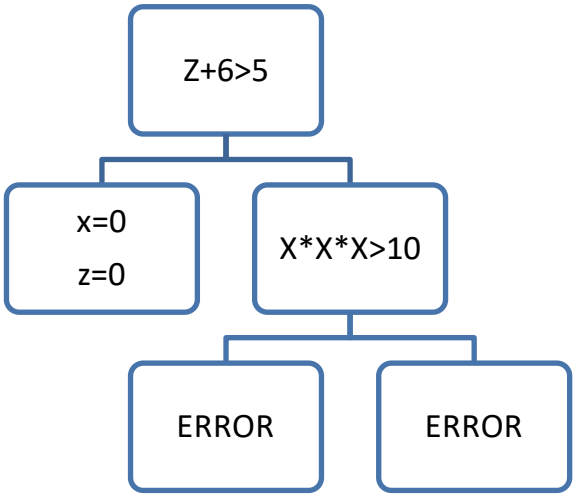
```
1:  testMe(int x, int y){
2:        if(y>5){
3:                assert(false);
4:        }else{
5:                if(x*x*x > 10){
6:                        assert(false);
7:                }
8:        }
9:  }
10: void main ( ){
11:       int x = symbolicinput();
12:       int z = symbolicinput();
13:       int y = z+6;
14:       testMe(x,y);
15: }
```

شکل ۲-۱ نمونه برنامه ساده

ابتدا اجرای نمادین برنامه شکل ۲-۱ را بررسی می‌کنیم. وقتی در اجرای نمادین متغیری نمادین در نظر گرفته می شود (خط ۱۱ و ۱۲) به این معنی است که آن متغیر نماینده تمام مقادیر ممکن برای آن نوع است. مثلا متغیر x نماینده تمام مقادیر ممکن برای نوع int است. این مقدار را با حرف بزرگ نشان خواهیم داد. مثلا مقدار نمادین متغیر x را با X نشان می‌دهیم. برنامه با مشخص شدن این مقادیر اجرا می‌شود. در خط ۱۳ برنامه متغیر y با عمل انتساب مقدار Z+6 را خواهد پذیرفت که

همان طور که مشخص است مقداری نمادین است. سپس در خط ۱۴ تابع testMe با مقادیر نمادین X و Z+6 فراخوانی می‌شود.

اجرای نمادین در مواجه با دستورات شرطی از قبیل if، شرط مربوط به آن را به صورت یک عبارت منطقی به عنوان شرط مسیر[۱] نگهداری می‌کند. همان طور که مشخص است دستورات شرطی موجود در برنامه موجب می شوند تا مجموعه د ستورهایی که قرار ا ست بعد از آن اجرا شوند، تصمیم‌گیری شوند. شرط مسیر عبارتی ا ست که از عطف[۲] شرط‌های دستورهای شرطی موجود در آن مسیر بدست می‌آید. شرط مسیر در ابتدا مقدار در ست[۳] دارد. برای مثال در خط ۲ برنامه، شرط $y > 5$ که معادل $Z+6 > 5$ ا ست به شرط م سیر ا ضافه می شود و داریم: PC=(Z+6>5). اگر این شرط برقرار با شد، خط ۳ اجرا می‌شود. برای اینکه پوشش کامل مسیرهای برنامه بدست آید، یعنی خط‌های ۵ و ۶ هم اجرا شود، شرط مسیر باید PC=(Z+6<=5) باشد. در اجرای نمادین تمام حالت‌های ممکن برای شرط مسیر در نظر گرفته می‌شود. مجموعه شرط‌های مسیر ممکن در کنار هم درخت اجرا[۴]ی بر نا مه را می سازند. در شکل ۲-۲ درخت اجرای برنامه نمونه را می‌بینید.



شکل ۲-۲ درخت اجرای اجرای نمادین برنامه نمونه

برای تولید موردآزمون برای هر مسیر، شرط مسیر مربوط به آن، به ابزار «حل کننده قید[۵]» داده می شود. این ابزارها با دریافت یک عبارت منطقی، م قادیر مت ناظر با هر متغیر در آن عبارت را پیدا می‌کنند که به ازای آنها کل

---

۱ Path Condition

۲ And

۳ True

۴ Execution Tree

۵ Constraint Solver

عبارت درست خواهد بود. مثلا به ازای x=0 و y=0 عبارت PC=Z+6>5 درست است و خط ۳ برنامه اجرا می‌شود. حل‌کننده‌های قید دارای توان محدودی هستند برای مثال توانایی حل عبارت‌های غیرخطی مثل X*X*X>10 را ندارند. در نتیجه با اجرای نمادین نمی‌توان موردآزمونی تولید کرد که به ازای آن خط ۶ برنامه اجرا شود.

در روش پویا-نمادین متغیرها علاوه بر شکل نمادین به صورت مقدار عینی[1] نیز در نظر گرفته می شوند. مقدار عینی در ابتدا به صورت دلخواه انتخاب می شود. برای مثال در این برنامه مقدار اولیه عینی x=1 و z=1 به صورت دلخواه انتخاب می‌شود. شرط مسیر استخراج شده با این ورودی‌ها PC=(Z+6>5) خواهد بود. برای تولید ورودی عینی جدید مکمل شرط مسیر یعنی PC=(Z+6<=5) به حل‌کننده قید داده می‌شود. مقدار جدید z=-2 و x=1 خواهد بود که با آن شرط مسیر PC=(Z+6<=5) and (x*x*x<10) تولید می شود. همان طور که گفته شد، حل‌کننده قید 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 z=-2 باعث می‌شود برنامه به خط ۶ برسد.

۱-۲- چالش‌های اجرای پویا-نمادین

اجرای پویا-نمادین با چالش‌های مختلفی روبه‌رو است که در ادامه گفته خواهند شد. هر یک از این چالش‌ها به نحوی سعی شده است که در کارهای گذشته حل شوند که در ادامه فصل بیان خواهند شد.

چالش‌های اجرای پویا-نمادین عبارتند از:

- **حافظه**: موتور اجرای نمادین چگونه اشاره‌گرها، آرایه‌ها و ساختمان داده‌ها را پردازش می‌کند؟ آیا می‌تواند ساختمان‌داده‌های پیاده‌سازی شده برنامه نویس را هم پردازش کند؟
- **محیط**: برنامه مورد آزمون ممکن است که با محیط خود و متغیرهای موجود در آن تعامل داشته باشد. مثلا برنامک‌های اندرویدی که پیوسته با سیستم عامل در ارتباطند و قطعه کدهای مختلف

[1] Concrete

موجود در آن را اجرا می‌کند. همچنین سیستم فایل، شبکه، تعاملات کاربر با برنامه و غیره هم از این دست هستند. موتور اجرای نمادین باید برای این موارد راه حل داشته باشد. پس اجرای نمادین در پلتفرم‌های مختلف متفاوت خواهد بود.

- **حلقه‌ها**: موتور اجرای نمادین باید در مورد تعداد دفعات اجرای بدنه یک حلقه تصمیم‌گیری کند. ممکن است یک حلقه شرط خاتمه نداشته باشد در نتیجه آزمون برنامه دچار انفجار مسیر خواهد شد.
- **انتخاب مسیر و مسئله انفجار مسیر**: انتخاب اینکه کدام یک از مسیرهای برنامه اجرا شود و هیوریستیک انتخاب کننده آن بسیار مهم است. برنامه‌های واقعی مسیرهای زیادی دارند که اجرای همه آنها موجب انفجار مسیر شده و هیچگاه فرایند آزمون تمام نمی‌شود.
- حل‌کننده‌های قید: حل‌کننده‌های قید محدودیت‌های زیادی دارند و نمی‌توانند همه قیدها را حل کنند. نیاز است تا با روش‌هایی این قیدها و تعداد آنها کاهش یابد و ساده شوند.
- **کدهای باینری**: در دنیای واقعی برنامه‌هایی وجود دارند که کد آنها در دسترس نیست و نیاز است تا این برنامه‌ها را با وجود کد باینری آزمود هر چند که وجود کد منبع و سطح بالای آنها تحلیل را آسان‌تر می‌کند.

۱-۲- انواع اجرای پویا-نمادین

در این حوزه اجرای پویا-نمادین به دو صورت آفلاین و آنلاین و یا ترکیب این دو صورت می‌گیرد. منظور از اجرای پویا-نمادین آفلاین است که در هر بار اجرا، یک مسیر انتخاب می‌شود این موضوع باعث می‌شود استفاده از حافظه کم باشد ولی تعداد زیادی از دستورات و کدها بارها به صورت تکراری اجرا می‌شوند. در اجرای آنلاین برخلاف آفلاین با یک بار اجرا تمام مسیرهای موجود اجرا می‌شوند. در این حالت با استفاده از دستور fork بر سر هر دستور شرطی، هر دو شاخه موجود همزمان اجرا می‌شوند. مزیت این روش در این است که هر دستور فقط یکبار اجرا می‌شود ولی استفاده از حافظه در آن به شدت زیاد است.

در برخی از کارها سعی شده است که از ترکیب این دو روش استفاده شود تا مزیت هرکدام را در خود داشته باشد. در این حالت «ترکیبی» تا زمانی که استفاده از حافظه به حد معین شده خود نرسیده

اســت، برنامه به صــورت آنلاین اجرا می‌شــود. بعد از آن اجرا آفلاین می‌شــود تا زمانی که به اندازه کافی حافظه آزاد شود تا دوباره اجرا به شکل آنلاین ادامه پیدا کند.

۱-۲- کارهای گذشته

در ادامه کارهایی را بیان خواهیم کرد که در مورد اجرای پویا-نمادین از سال ۲۰۰۵ تا کنون انجام شده اســت. در مورد هر کار ویژگی‌های خاص آن و بهبودی که در مورد هر چالش داشــته اســت را توضــیح خواهیم داد. در جدول1 کارهای گذشته شاخص آمده‌اند.

جدول1 کارهای گذشته

	سال	ابزار	زبان برنامه مورد آزمون	پلتفرم	نوع	ویژگی
۱	۲۰۰۵	DART	C	کامپیوتر شخصی	آفلاین	عمق‌اول
۲	۲۰۰۵	CUTE	C	کامپیوتر شخصی	آفلاین	مدل‌سازی حافظه، عمق‌اول کراندار، بهینه‌سازی
۳	۲۰۰۶	jCUTE	جاوا	کامپیوتر شخصی	آفلاین	CUTE، همروندی
۴	۲۰۰۶	EXE	C	کامپیوتر شخصی	آنلاین	مدل‌سازی حافظه، عمق‌اول و سطح‌اول ترکیبی، بهینه‌سازی، STP
۵	۲۰۰۷	Hybrid	C	کامپیوتر شخصی	آفلاین	CUTE، ترکیب اجرای دلخواه و پویا-نمادین
۶	۲۰۰۷	Compositional	C	کامپیوتر شخصی	آفلاین	DART، ترکیب Compositional و پویا-نمادین

۷	KLEE	C	کامپیوتر شخصی	آنلاین	EXE، بهینه‌سازی انتخاب مسیر و حل‌کننده قید، متن‌باز
۸	jFUZZ	جاوا	کامپیوتر شخصی	آفلاین	متن‌باز، بهینه‌سازی کارهای گذشته
۹	SPF	جاوا	کامپیوتر شخصی	آفلاین	ترکیب وارسی مدل و اجرای نمادین
۱۰	LCT	جاوا	کامپیوتر شخصی	آفلاین	متن‌باز، SMT، معماری برنامه نویسی سوکت
۱۱	AEG	باینری و C	کامپیوتر شخصی	آفلاین	آسیب‌پذیری سرریز بافر، خاصیت ایمنی
۱۲	ACTEVE	اندروید	Phone	آفلاین	متن‌باز، رخدادمحور
۱۳	MAYHEM	باینری	کامپیوتر شخصی	ترکیبی	آسیب‌پذیری سرریز بافر، قالب رشته، مدل‌سازی‌حافظه
۱۴	Jalangi	جاوا اسکریپت	وب	آفلاین	ثبت-بازاجرای انتخابی، مقادیر سایه
۱۵	AppIntent	اندروید	Phone	آفلاین	کشف نشت حریم خصوصی
۱۶	SIG-Droid	اندروید	Phone	آفلاین	استفاده از کلاس‌های Mock، گراف فراخوانی توابع
۱۷	Condroid	اندروید	Phone	آفلاین	پویا-نمادین Call Flow + Graph

۱۸	۲۰۱۶	Driller	باینری	کامپیوتر شخصی	ترکیبی	ترکیب instrumented- Genetic-Fuzzer با پویا-نمادین

در ادامه به توضیحی مختصر در رابطه با هر مقاله می‌پردازیم:

۱. در سال ۲۰۰۵ اولین ابزار با روش پویا-نمادین آفلاین به نام Dart [۱] ارائه شد. این ابزار از lp solve به عنوان حل‌کننده قید استفاده می‌کند. همچنین محدود به زبان C است و مدل سازی حافظه ندارد. علاوه بر آن از برنامه‌های هم‌روندی پشتیبانی نمی‌کند. از جست‌وجوی DFS برای انتخاب مسیرها در درخت اجرا استفاده می‌کند و بهینه سازی برای ارسال قیدها به حل کننده قید ندارد همچنین این ابزار در حل قیدهای مربوط به اشاره‌گرها مشکل دارد.

۲. در سال ۲۰۰۵، ابزار CUTE [۲] با روش پویا-نمادین آفلاین ارائه شد که از lpsolve استفاده می‌کند. این ابزار هم محدود به زبان C است و از هم‌روندی پشتیبانی نمی‌کند. ولی مدل سازی حافظه دارد و از نگاشت منطقی ورودی‌ها استفاده می‌کند و مشکل قیدهای اشاره‌گر را حل کرده است. همچنین از جست‌وجوی DFS کراندار برای انتخاب مسیرها استفاده می‌کند و بهینه‌سازی برای ارسال قیدها به حل کننده قید دارد. روش‌های بهینه سازی آن عبارتند از: برر سی سریع ارضاناپذیری، حذف قیدهای معمول و حل افزایشی.

۳. در سال ۲۰۰۶، ابزار JCUTE [۳] با روش پویا-نمادین آفلاین، ارائه شد که از lpsolve استفاده می‌کند. این ابزار محدود به زبان جاوا اســـت ولی مدل‌ســـازی حافظه دارد و مانند CUTE از نگاشت منطقی ورودی‌ها استفاده می‌کند. همچنین از هم‌روندی پشتیبانی می‌کند یعنی علاوه بر ورودی‌های برنامه، زمانبند نخ‌ها هم باید به صـــورت خودکار برنامه‌ریزی شـــود. این ابزار از جســـت‌وجوی DFS برای انتخاب مســـیرها اســـتفاده می‌کند و مانند CUTE بهینه‌ســـازی برای ارسال قیدها به حل کننده قید دارد.

۴. در ســـال ۲۰۰۶، ابزار EXE [۴] با روش پویا-نمادین آنلاین، ارایه شـــد که از STP اســـتفاده می‌کند. این ابزار محدود به زبان C اســـت و از هم‌روندی پشـــتیبانی نمی‌کند. ولی مدل‌ســـازی حافظه دارد. حافظه را مجموعه‌ای از بایت های بدون نوع در نظر می‌گیرد. همچنین از جست‌وجوی DFS و BFS به صورت ترکیبی برای انتخاب مسیرها استفاده می‌کند. علاوه برآن

بهینه سازی برای ار سال قیدها به حل کننده قید دارد. ایده‌های این ابزار در این مورد ا ستفاده از روش کش و شناسایی زیرقیدهای مستقل و حذف زیرقیدهای بی ارتباط است.

۵. اجرای هیبرید [۵] به صـــورت ترکیبی اجرای دلخواه[۱] و پویا-نمادین را انجام می‌دهد تا بتواند از مزیت‌های هر یک استفاده کند. کار ارائه شده بروی ابزار CUTE است. ابتدا کد به صورت عینی اجرا می‌شـــود. هر گاه اجرا اشـــباع شـــد اجرا به پویا-نمادین تغییر میابد تا بتواند به صـــورت عمق‌محدود به پوشش بیشتری از کد برسد. دوباره بعد از یافتن مسیر جدید اجرا به عینی تغییر میابد. اجرای هیبرید برای برنامه‌های تعاملی مثل برنامه‌های رخدادمحور یا دارای GUI مناسـب اســت. این اجرا همان محدودیت‌های اجرای پویا-نمادین را دارد. ممکن اســت به پوشـــش ۱۰۰ در صد از کد نر سد ولی از نظر نوی سندگان پو شش کامل ن شانه ای برای قابل اعتماد بودن[۲] کد نیست.

۶. کار مورد شـــش در جدول [۶]، از DART به عنوان موتور پویا-نمادین اســـتفاده می کند. هدف این کار توسعه DART برای برنامه های واقعی با تعداد خط کد بالاست به همین دلیل از تحلیل ایستای Compositional استفاده می کند که برای توابع function summery استخراج می کند و به جای اجرای هر باره یک تابع از summery آن استفاده می کند و آن را به شرط مسیر اضافه می‌کند.

۷. ابزار KLEE [۷] در ســـال ۲۰۰۸، با روش پویا-نمادین آنلاین ارائه شـــد که از STP اســـتفاده می‌کند. این ابزار برای آزمون برنامه‌های واقعی محدود به زبان C است. مدل سازی محیط اجرای برنامه(ســیســتم فایل) و مدل‌ســازی حافظه دارد. حافظه را مجموعه‌ای از بایت‌های بدون نوع در نظر می‌گیرد. ولی از هم‌روندی پشـــتیبانی نمی‌کند. این ابزار روش‌های انتخاب دلخواه و انتخاب برای پوشـــش بیشـــترین مســـیرها را به صـــورت ترکیبی اســـتفاده می‌کند. بر اســـاس یک ســـری هیوریســـتیک به حالت‌ها وزن اختصـــاص داده می‌شـــود و ســـپس به صـــورت دلخواه یکی از این حالت‌ها انتخاب می‌شـــوند. در حالت دوم، هیوریســـتیک‌ها بر اســـاس کمترین فاصـــله تا دســـتور پوشش داده نشده، بیشینه فراخوانی حالت و یا اینکه یک حالت اخیرا دستور جدیدی را پوشش

---

[۱] Random

[۲] Reliability

داده ا ست یا نه، محا سبه می شود. ترکیب این دو ا ستراتژی باعث می شود هم پو شش تمامی دستورات فراهم شود و هم از گیر کردن در حلقه جلوگیری به عمل آید. برای بهینه سازی قیدها به حل کننده قید از روش‌هایی مثل روش کش اســـتفاده می‌کند. این ابزار گســـترش یافته ابزار EXE است.

۸. jFuzz [] ابزار متن باز برای جاوا ست. نوآوری خا صی ندارد و ترکیب بهینه سازی های کارهای قبلی مثل KLEE، CUTE و غیره را در خود دارد. این ابزار بروی پروژه JPF [] پیاده ســـازی شده است.

۹. برای اجرای نمادین برنامه‌های به زبان جاوا، ابزار SPF [] 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 پشـــتیبانی می‌کند که با اســـتفاده از آنها می‌توان قیدهای مختلف را تحلیل کرد. به طور خاص برای رشـــته‌ها که در تحلیل ما بســـیار 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 پژوهش با تغییر SPF به دنبال تشـــخیص آســـیب‌پذیری تزریق به برنامک‌های اندرویدی هستیم.

۱۰. ابزار LCT [۸] ابزار متن باز روی جاوا است. در این ابزار سعی شده از معماری کارگذار-کارخواه[۱] برای ارتباط بین حل‌کننده قید و تحلیلگر استفاده کند. مشکل این ابزار این است که از چندنخی پشتیبانی نمی‌کند و توانایی پیدا کردن خطاهایی مثل کد روبه‌رو را ندارد. a[j]=1; if(a[i]!=0) ERROR;.

۱۱. تحلیل کد برنامه، به تنهایی برای تحلیل کافی نیســـت. چون کد برنامه اطلاعی از مقادیر و چینیش داده ها در زمان اجرا ندارد. در مقابل تحلیل باینری مقیاس‌پذیر نیست و مفاهیمی مثل متغیرها ساختمان داده ها(آرایه ها و ..) در آن معنی ندارد. تنها با فریم های پشته و دستورات پرش و آدرس های حافظه ســـر و کار دارد. در AEG [۹] از ترکیب هر دو روش یعنی تحلیل باینری و کد برنامه استفاده شده است. نحوه کار AEG به این صورت است:

---

[۱] client-server

- ابتدا با استفاده از کد تحلیل نمادین صورت می گیرد تا به دستور آسیب‌پذیر برسد.
- سپس شرط مسیر به حل کننده قید داده می شود تا ورودی مناسب تولید شود.
- سپس به صورت پویا و با استفاده از ورودی تولید شده، فایل باینری برنامه تحلیل می شود تا اطلاعات زمان‌اجرا[۱] یعنی ساختار حافظه مثل آدرس بافر سرریز شده و آدرس بازگشت استخراج شود.
- AEG قیدهای جدیدی مربوط به اطلاعات ساختار حافظه تولید می کند و به شرط مسیر اضافه می کند. این قیدها باید شامل shell code و آدرس بازگشت به shell code باشند. سپس شرط مسیر به حل کننده قید داده می شود تا ورودی مناسب تولید شود.
- در نهایت AEG ورودی تولید شده را به برنامه می دهد تا بررسی کند که کد بهرجو[۲] آیا اجرا می شود یا نه! اگر حل کننده قید نتواند شرط مسیر را حل کند، AEG آن را رها می کند و فرایند را ادامه می دهد.

۱۲. ابزار ACTEVE [۱۰] اجرای پویا-نمادین آفلاین برای برنامه‌های گوشی همراه است که از Z3 SMT solver استفاده می‌کند. این ابزار برای برنامک‌های اندرویدی[۳] ارائه شده است. این برنامک‌ها رخدادمحور[۴]، هستند. منظور از رخدادمحور بودن این است که کاربر با برنامه تعامل دارد و رفتار او در فرایند اجرای برنامه موثر است. یعنی علاوه بر داده‌ها، رخدادها هم مسیر اجرای برنامه را تعیین می‌کنند. چالش این آزمون تولید یک رخداد و همچنین تولید ترتیبی از رخدادها است.

---

۱ Runtime

۲ Exploit

۳ Android Apps

۴ Event Driven

در این مقاله از روش پویا-نمادین برای تولید رخدادها ا ستفاده می شود. برای این منظور SDK[۱] و برنامه تحت آزمون باید تجهیز[۲] شـوند. سـپس در حین اجرای یک رخداد عینی، یک رخداد به صورت نمادین هم تولید می‌شود که تمام قیدهای مسیر را در خود نگهداری می‌کند. با این روش برای ترتیبی از رخدادها باید همه حالت های وقوع رخدادها بررسـی شـود. (دوتایی، سـه‌تایی، چهارتایی و ...) و جای‌گشت‌های مختلف رخدادها در هر ترتیب نیز در نظر گرفته شود که فضای حالت خیلی بزرگی دارد. این کار محدود به رخداد ضربه[۳] ا ست. علاوه بر آن همان طور که گفته شد، نیاز به بهینه سازی برای کاهش فضای حالت در ترتیب‌های مختلف از رخدادها دارد. در این کار با حذف ویجت[۴]های غیرفعال، حذف ویجت‌های بدون کنش مثل LinearLayout و محدود کردن آزمون به رخدادهایی که در برنامه استفاده می‌شود، تا حدودی این بهینه‌سازی انجام شده است. ACTEVE مدل‌سازی حافظه ندارد و از هم‌روندی هم پشتیبانی نمی‌کند.

۱۳. مراحلی که MAYHEM [۱۱] برای تولید اکسپلویت طی می‌کند:

- ابزار MAYHEM با تعریف یک پورت کار خود را شروع می‌کند. و کدهای آسیب‌پذیر را از همین طریق دریافت می‌کند. این موضوع باعث می‌شود که ابزار بداند چه کدهایی در اختیار مهاجم است.
- واحد CEC[۵] برنامه آسیب‌پذیر را دریافت می‌کند. به SES[۶] وصل می‌شود تا مقداردهی‌های اولیه صورت پذیرد. سپس کد به صورت عددی اجرا می‌شود و همزمان تحلیل آلایش[۷] پویا نیز روی آن اجرا می‌شود.

---

۱ Software Development Kit

۲ Instrument

۳ Tap Event

۴ Widget

۵ Concrete Execution Client

۶ Symbolic Execution Server

۷ Taint Analysis

- اگر CEC با یک بلاک کد آلوده یا یک پرش آلوده رو به رو شود. (منظور جایی است که لازم است تا از کاربر ورودی دریافت شود)، CEC موقتاً اجرا نمی شود و شاخه آلوده به SES برای اجرای نمادین ارسال می‌شود. SES مشخص می‌کند که آیا اجرای شاخه ممکن هست یا نه!
- واحد SES به صورت موازی با CEC اجرا می‌شود و بلاک‌های کد را دریافت می‌کند. این بلاک‌ها به زبان میانی تبدیل می‌شوند و به صورت پویا-نمادین اجرا می‌شود. مقادیر عددی مورد نیاز از CEC دریافت می‌شود.
    - فرمول قابلیت اکسپلویت مشخص می‌کند که:
        - آیا مهاجم می‌تواند کنترل اجرای دستورات یا
        - اجرای PAYLOAD را بدست آورد یا نه؟
- وقتی به یک پرش آلوده می‌رسد SES تصمیم می‌گیرد که آیا FORK لازم هست یا نه. اگر باشد اجراهای جدید اولویت بندی شده و یکی اجرا می‌شود. اگر منابع تمام شوند SES رویه بازگشت را اجرا می‌کند. در نهایت بعد از اتمام اجرای یک پردازه تعدادی موردآزمون تولید می‌شوند.
- در پرش‌های آلوده یک فرمول بهره‌جو[۱] تولید و به SES داده می‌شود اگر قابل ارضا بود یعنی کد از این مسیر آسیب‌پذیر است.
- ابزار Jalangi [۱۲] در سال ۲۰۱۳ با اجرای پویا-نمادین آفلاین ارائه شد. این ابزار محدود به زبان جاوا اسکریپت است ولی مدل‌سازی حافظه و بهینه‌سازی برای انتخاب مسیر اجرای برنامه ندارد همچنین از هم‌روندی پشتیبانی نمی‌کند. این ابزار از ثبت-بازاجرای انتخابی[۲] استفاده می‌کند. برنامه‌های به زبان جاوا اسکریپت ممکن هست از کتابخانه‌های مختلفی مثل jQuery استفاده کند. Jalangi این ویژگی را دارد که کاربر می‌تواند انتخاب کند که رفتار کتابخانه‌ای خاص، تنها بررسی و تحلیل شود. Jalangi همچنین از مقادیر سایه[۳] استفاده می‌کند. این مقادیر اطلاعاتی

---

۱ Exploit

۲ Selective Record-Replay

۳ Shadow Values

اضافی (مثل آلایش شدن یا نمایش نمادین) را در مورد داده‌های اصلی در خود نگهداری می‌کنند. از این مقادیر در اجرای نمادین یا تحلیل آلایش استفاده می‌شود.

۱۴. ایده اصلی ابزار APPINTENT [۱۳] استفاده از اجرای نمادین برای به دست آوردن دنباله رویدادهایی است که موجب یک انتقال داده مشخص درون گوشی همراه شده‌اند. اما اجرای نمادین در کنار مزایای قابل توجه‌ای که در اختیار می‌گذارد از نظر مصرف حافظه و زمان بسیار ناکارآمد است. نوآوری علمی ابزار APPINTENT ارائه بهبودی برای اجرای نمادین با کاهش فضای جست‌وجو در برنامک‌های اندرویدی و بدون از دست رفتن پوشش کد بالا است. در ابزار APPINTENT از تحلیل آلایش ایستا استفاده شده است که با استفاده از آن تمامی انتقال داده‌های حساس و دنباله رویدادهای مربوط به آنها استخراج می شود. در ادامه با اجرای نمادین هدایت‌شده توسط اطلاعات به دست آمده از تحلیل آلایش ایستا، ورودی‌های حساس برای برنامه تولید می‌شود. پوشش کد کافی نیز بنابر ماهیت ذاتی اجرای نمادین به دست می‌آید.

۱۵. در سال ۲۰۱۵ ابزار Sig-Droid [۱۴] برای آزمون برنامک‌های اندرویدی ارائه شده است. در این ابزار سعی شده است برنامک‌ها روی JVM[۱] 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. در این ابزار نقطه شروع برنامه[۲] از طریق تحلیل ایستا و گراف فراخوانی توابع بدست می‌آید. کلاس‌های SDK و وابستگی‌های به آن به وسیله کلاس Mock حل شده است و در نهایت با اجرای نمادین کد روی SPF سعی شده است تمام مسیرهای موجود در برنامه پوشش داده شوند.

۱۶. ابزار Condroid [۱۵] در سال ۲۰۱۵ با گسترش ابزار ACTEVE ارائه شده است. در این ابزار با استفاده از تحلیل ایستا و گراف کنترل جریان نقطه شروع به برنامه را استخراج می‌کند. در این ابزار با یافتن نقاط حساس در کد، مثلا تعداد زیاد دستورات شرطی پشت سرهم، سعی می‌کند

---

[۱] Java Virtual Machine

[۲] Main Method

بمب منطقی را در برنامک‌های اندرویدی تشـخیص دهد. این ابزار همان مشـکلات ACTEVE یعنی انفجار مسیر را به ارث برده است.

۱۷.ابزار Driller [۱۶] از ۴ قسمت اصلی تشکیل شده است:

- موردآزمون به عنوان ورودی: ابزار به صورت خودکار توانایی تولید موردآزمون را دارد ولی ورودی آن توسط کاربر می‌تواند به ابزار سرعت بخشد.
- Fuzzing: ابزار ابتدا با Fuzzing شـروع به کار می‌کند. اگر به ورودی‌های «مشـخص» برسـد fuzzer گیر می‌کند.
- اجرای پویا-نمادین : وقتی fuzzer گیر کرد اجرای پویا-نمادین شروع به کار می‌کند تا مسیر جدیدی را پیدا کند.
- Repeat: وقتی مسـیر جدید پیدا شـد، اجرا دوباره بهfuzzer سـپرده می‌شـود و اجرا ادامه پیدا می‌کند.

یک ویژگی مهم Driller است که وقتی اجرا به موتور پویا-نمادین داده می شود، اجرای پویا-نمادین دچار انفجار مسیر نخواهد شد. چون که fuzzer مسیر اجرای پیشین خود را به موتور پویا-نمادین می‌دهد و اجرای پویا-نمادین تنها سـعی می‌کند با not کردن یکی از شـرطهای مسـیر ورودی از fuzzer به مسیری جدیدی برسد.

فصل سوم

تشخیص آسیب‌پذیری

تشخیص آسیب‌پذیری

فصل چهارم

راه‌کار پیشنهادی

راه‌کار پیشنهادی

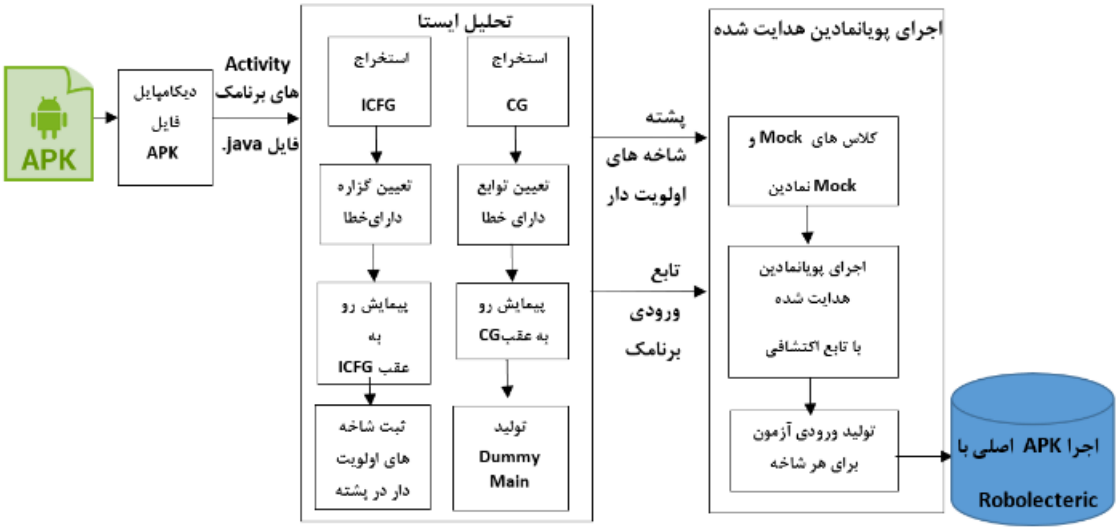
در فصل ... چالش‌های مربوط به تحلیل برنامک‌های اندرویدی توضیح داده شد. در این فصل ما برای مقابله با هر یک و بهینه شدن فرایند تحلیل ایده‌های خود را ارائه خواهیم داد. همچنین در این پژوهش ما بر روی دو موضوع کار کرده‌ایم:

- ارائه یک هیوریستیک برای اجرای پویا-نمادین هدایت شده
- تشخیص آسیب‌پذیری تزریق

هر دوی این موضوع‌ها دارای مراحل مشترکی هستند. به ترتیب هر کدام از آنها توضیح داده خواهند شد. در موضوع اول به صورت کلی‌تر به دنبال یافتن خطا در برنامک‌های اندرویدی هستیم و برای بهتر شدن فرایند کشف خطا، یک هیوریستیک ارائه خواهیم داد. در موضوع دوم با کمک گرفتن از ایده‌های موضوع اول و اضافه کردن یک سری ایده و الگوریتم، تشخیص آسیب‌پذیری تزریق در برنامک‌های اندرویدی را ارائه خواهیم داد.

۱-۲- ارائه یک هیوریستیک برای اجرای پویا-نمادین هدایت شده

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 برنامه و پشته حاوی شاخه‌های اولویت‌دار است. با ا ستفاده از خروجی تحلیل ایستا، اجرای پویا-نمادین همراه با هیوریستیک ارائه شده اعمال می گردد،



شکل ۳ معماری کلی طرح پیشنهادی

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.

۱-۳-۲- دیکامپایل برنامک

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

۲-۳-۲- تحلیل ایستا

در کار ما تحلیل ایستا به دو منظور انجام می‌شود. از آنجایی که موتور اجرای پویا-نمادین برای برنامک‌های اندرویدی وجود ندارد، ما از موتور SPF استفاده خواهیم کرد. این موتور برای برنامه‌های جاوا تولید شده است. برنامه‌های به زبان جاوا نقطه شروع مشخص به برنامه دارند ولی برنامک‌های اندرویدی این گونه نیستند. در بخش تحلیل ایستا ابتدا نقطه ورودی برنامک را با تحلیل «گراف فراخوانی توابع[1]» استخراج می‌نماییم. برای اینکه تحلیل‌های ما دقیق‌تر و با سربار کمتر صورت پذیرد، تحلیل ایستای دیگری نیز علاوه بر مورد اول صورت می‌پذیرد. در این تحلیل با پیمایش روبه‌عقب «گراف کنترل جریان بین تابعی[2]»، پشته شاخه‌های اولویت‌دار را تعیین می‌کنیم. این پشته در اجرای هدایت شده پویا-نمادین به ما کمک خواهد کرد. هر یک از این دو مورد در ادامه شرح داده خواهند شد.

۱-۱-۱-۱- استخراج نقطه ورودی برنامه

برنامک‌های اندروید برخلاف برنامه‌های دیگر نقطه شروع مشخصی ندارند. درواقع 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. مثلا یک برنامک با آمدن یک رخداد مثل دریافت یک پیامک ممکن است کار خود را شروع کند یا همان برنامک با باز کردن عادی آن و مثلا فشردن یک دکمه کار خود را آغاز می‌کند.

[1] Call Graph (CG)

[2] Inter- Control Flow Graph (ICFG)

در اجرای پویا-نمادین با SPF ما نیاز داریم تا از یک نقطه شروع مشخص کار را آغاز کنیم چون که SPF برای برنامه‌های به زبان جاوا نوشته شده است. برنامه‌های به زبان جاوا دارای تابع شروع (main) هستند و SPF کار را از همان تابع شروع می‌کند. به همین دلیل ابتدا گراف فراخوانی توابع برنامک را استخراج می‌کنیم. در این قسمت از 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.

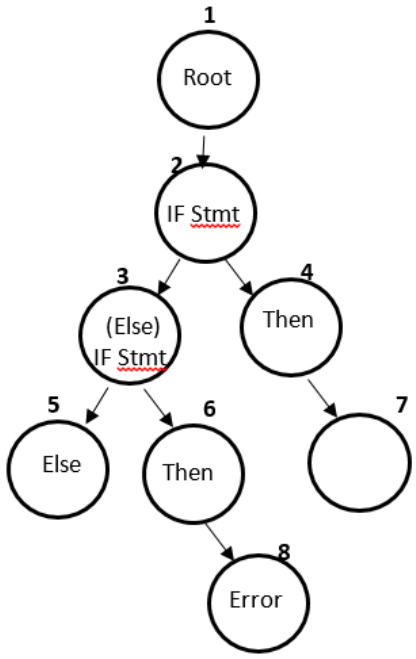
استثنای زمان‌اجرا می‌تواند از جنس «خطای تقسیم بر صفر»، «استثنای نقض محدوده آرایه» یا موارد دیگری باشد که در زمان اجرای برنامک اعلام[11] می‌شود. در برنامک‌های مورد آزمون، در نقاط مناسب برنامک، کد تولیدکننده این استثنا را قرار می‌دهیم.

برای تولید تابع نقطه ورودی به برنامه 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 [17] به دست آورده‌ایم. سپس الگورتیم پیمایش رو به عقب پیشنهادی خودمان را روی گراف استخراج شده اعمال کرده‌ایم. نمونه تابع نقطه شروع به برنامک در شکل ۴ دیده می‌شود.

```
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 برای برنامه MunchLife

۲-۱-۱-۱- تعیین پشته شاخه‌های اولویت دار



شکل ۵ مثالی از گراف کنترل جریان بین تابعی

برای به دســـت آوردن اولویت اجرای هر شـــاخه، 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 ریشـــه در گراف را به صـــورت رو به عقب پیمایش می‌کنیم و در این حین اطلاعات مربوط به انتخاب شـــاخه‌های مختلف در گراف را در یک پشـــته ذخیره می‌کنیم. در این نوشتار ما این پشته را «پشته شاخه‌های اولویت‌دار» می‌نامیم. این اطلاعات شـامل دسـتور شـرطی مورد نظر و اولویت شاخه‌های then و else نسبت به هم است. 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 شـــاخهthen بر else اولویت دارد را در پشـــته ذخیره می‌کنیم. در این مثال در گره ۳، دستور شرطی و اولویت شاخهthen بر else را به پشته اضافه می‌کنیم. همچنین برای گره ۲، دستور شرطی و اولویت else بر then را به پشته اضافه خواهیم کرد.

۳-۳-۲- تولید کلاس‌های Mock و Mock نمادین

برای اینکه برنامک روی JVM قابل اجرا باشـــد و چالش رخدادمحور بودن را حل کنیم، از کلاس‌های Mock به جای کلاس‌های اصـــلی SDK اســـتفاده کرده‌ایم. چالش‌های گفته شـــده در فصـــل ... 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 اســـت 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 در ستی ورودی‌های که قرار ا ست کاربر وارد کند، بعد از اجرای پویا-نمادین بد ست بیایند. برای مثال کلاس Mock مرتبط با EditText در شکل ۶ آمده است. همان طور که در شکل دیده می‌شود خطوطی

```
package android.widget;
import android.app.Activity;
import android.view.View;
import gov.nasa.jpf.symbc.Debug;
public class EditText extends View {
        String content;
        public EditText(String id) {this.content = Debug.makeSymbolicString(id);}
        public Object getText(){return content;}
        public void setOnKeyListener(OnKeyListener keyL) {}
        public void setOnFocusChangeListener (OnFocusChangeListener
onFocusChangeListener){}
        public Object getWindowToken(){return null;}
        public void requestFocus{      }
        public void setText(String text) {this.content=text;}
        public void clearFocus(){      }
        public void addTextChangedListener(Activity a){    }
        public void setError(Object object) {          }
}
```

شکل ۶ نمونه کلاس Mock نمادین تولید شده برای دریافت ورودی نمادین.

که پررنگ نشان داده شده‌اند، موجب شده‌اند که کلاس EditText به صورت Mock نماین تولید شود. ر شته Content در تابع سازنده[1] به صورت ر شته نمادین مقداردهی اولیه می شود. همین مقدار نمادین در تابع getText باز می‌گردد. همان طور که دیده می‌شـــود با ایده Mock نمادین لازم نیســـت تا تعامل

[1] Constructor

کاربر برای وارد کردن ورودی نمادین وجود داشـــته باشـــد و این موضـــوع چالش رخدادمحور بودن در برنامک‌های اندرویدی را حل می‌کند.

۴-۳-۲- 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 شـــاخه‌های اجرایی برنامه به طور نمایی زیاد می شود. 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 قسمت از پژوهش ما یک هیوریستیک را معرفی می‌کنیم که از انفجار مسیر در اجرای پویا-نمادین برنامک‌های اندرویدی جلوگیری می‌کند. در این هیوریســـتیک راهکار پیشـــنهادی ما بر دو ایده اســـتوار است:

الف) اجرای پویا-نمادین برنامک را به تابع‌های نقطه شـــروعی که به خطا منتهی می‌شـــوند، محدود می‌کنیم.

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

ایده الف را در بخش ۱-۱-۱-۱-اســـتخراج نقطه ورودی برنامه۱-۱-۱-۱- و ایده ب را در بخش تعیین پشته شاخه‌های اولویت‌دار شرح داده‌ایم. برای اجرای پویا-نمادین از SPF استفاده کرده‌ایم. در SPF به صـــورت پیش فرض درخت اجرا و کد برنامه پیمایش عمق‌اول می‌شـــود و هیچ اولویت‌گذاری روی انتخاب شـــاخه‌های مختلف وجود ندارد. این موضـــوع ممکن اســـت باعث شـــود که خطا در آخرین پیمایش و در آخرین شاخه اجرا شده در درخت اجرا کشف شود. در این قسمت از پژوهش برای بهبود این موضوع، ما از تحلیل ایستا استفاده کرده‌ایم. ما از تحلیل ایستا خودمان نقطه شروع به برنامک (بخش استخراج نقطه ورودی برنامه) و پشته شاخه‌های اولویت‌دار (بخش تعیین پشته شاخه‌های اولویت‌دار) 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 کدامیک از شاخه‌های پیش‌رو بدهیم. استفاده از این ایده باعث می‌شود که ابتدا مسیر منتهی به خطا زودتر اجرا شود. در SPF بر سر هر دستور شرطی تابعی به نام choiceGenerator وجود دارد. این تابع در اولین برخورد با یک دستور شرطی مشخص می‌کند که اولویت اجرا با شاخه then یا else است. به صورت پیش‌فرض این تابع همیشه شاخه else را انتخاب می‌کند و در نتیجه اجرا به صورت عمق‌اول خواهد بود. ما این تابع را بازنویسی کرده‌ایم. در این تابع با استفاده از اطلاعات موجود در پشته شاخه‌های اولویت‌دار، اولویت اجرای هر یک از شاخه‌ها را یافته و اعمال می‌کنیم. دلیل اینکه در تحلیل ایستا داده‌ها را در پشته ذخیره کرده بوده‌ایم این است که در تحلیل ایستا گراف مربوطه را به صورت روبه‌عقب پیمایش می‌کنیم ولی در SPF و اجرای پویا-نمادین درخت اجرا به صورت روبه‌جلو پیمایش می‌شود. پس اطلاعات دستور شرطی در سر پشته، اطلاعات مربوط به اولین دستور شرطی است که در اجرای پویا-نمادین با آن روبه‌رو خواهیم شد. با استفاده از اجرای پویا-نمادین در نهایت ورودی‌های آزمون مرتبط با خطاهای موجود در برنامک استخراج خواهند شد.

۵-۳-۲- 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 [18] استفاده کرده‌ایم. این ابزار یک ابزار آزمون واحد برنامک‌های اندرویدی است. با این ابزار می‌توان قسمتی از برنامک را با دادن ورودی‌های مناسب و فراخوانی تابع‌هایی که در اجرای آن مسیر خاص از برنامه لازم هستند، آزمود. پیش از این برای اجرای پویا-نماین در SPF لازم بود تا نقطه شروع به برنامک تولید شود. با استفاده از اطلاعات استخراج شده در آن مرحله، ورودی مناسب به ابزار Robolectric را تولید می‌کنیم. نمونه‌ای از آن در شکل ۷ آمده است. همان طور که دیده می‌شود خط ۳ از شکل ۴ مشابه خط ۲ از شکل ۷ است. خطوط۴ و ۵ از شکل ۴ به صورت خودکار در ابزار اجرا می‌شوند.

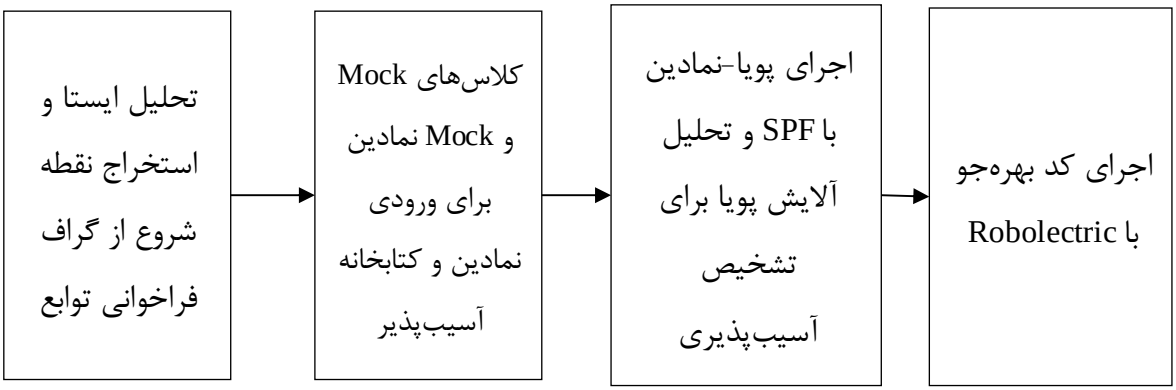
```
1: public void TestofApp() throws Exception {
2:        Activity ma = Robolectric.setupActivity(MunchLifeActivity.class);
3:}
```

شکل ۷ نمونه کد آزمون در Robolectric برای برنامه MunchLife.

۱-۲- اجرای پویا-نمادین برای تشخیص آسیب‌پذیری تزریق

در شکل ۸ نمای کلی از مراحل پیشنهادی برای تشخیص آسیب‌پذیری تزریق در برنامک‌های اندروید آورده شده است. برای انجام کار قسمت‌هایی از معماری که در قسمت قبل توضیح دادیم را در اینجا استفاده کرده‌ایم. در شکل ۸ به صورت کلی این قسمت‌ها آمده است. در ادامه این فرایند به طور کامل توضیح داده خواهد شد.

در این قسمت از 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 می شود 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



شکل ۸ فرایند تشخیص آسیب‌پذیری در ابزار

متغیرها در آن و خروجی تابع‌های آن نمادین باشند. سپس برنامک را به شکل پویا-نمادین اجرا می‌کنیم. در حین اجرا، وضـعیت نمادین‌بودن متغیرها را ذخیره می‌کنیم. برای مثال در str = str1+str2; (جمع دو رشـته) که str1 نمادین و str2 عینی اسـت، بعد از اجرای دسـتور، str را نمادین در نظر می‌گیریم. هرگاه اجرا به تابع آسـیب‌پذیر برسـد، با توجه به اطلاعات ذخیره‌شـده 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. 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 آسـیب‌پذیر نیز بایسـتی تولید کرد. کلاس‌های آسـیب‌پذیر هم کلاس‌هایی از SDK هسـتند و باید به شـکل Mock نمادین آنها را تولید کنیم. اجرای پویا-نمادین ادامه پیدا می‌کند تا زمانی که به تابع نشـت برسـیم، 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.

بر اسـاس آخرین دانسـته ما تاکنون، ابزاری برای تشـخیص آسـیب‌پذیری در برنامک‌های اندرویدی با اجرای پویا-نمادین ارائه نشـده اسـت. در منبع [۶] عنوان شـده اسـت که آسـیب‌پذیری 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 با شد. 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 داده‌ایم.

۱-۲- 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 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 ۹)

```
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: }
```

شکل ۹ نمونه dummyMain تولید شده برای اجرا در SPF

۱-۲- تولید کلاس‌های Mock و Mock نمادین

در این کار ما به دو منظور از کلاس‌های Mock اســـتفاده کرده‌ایم. برای اینکه برنامه روی JVM اجرا کنیم و مســئله رخدادمحور بودن را حل کنیم، از کلاس‌های Mock به جای کلاس‌های SDK اســتفاده کردیم. همچنین اگر جایی نیاز به رخدادی توســط کاربر باشــد (مثل فشــردن دکمه) آن رخداد را در dummyMain با فراخوانی تابع Mock مرتبط با آن (مثلا performClick) شــبیه‌ســازی می‌کنیم. همچنین برای حل مسـئله انفجار مسـیر، تنها کلاس‌هایی که از تحلیل ایسـتا اسـتخراج کردیم را تحلیل می‌کنیم و بقیه کلاس های برنامک را Mock می‌کنیم. در هر دوی این حالت ها وجود کلاس Mock هزینه و سربار تحلیل را کاهش می‌دهد.

همان‌طور که گفته شـد، برای اینکه تحلیل آلایش ما به شـکل کامل صـورت پذیرد، باید کلاس مربوط به تابع آســیب‌پذیر و تابع منبع را به شــکل Mock نمادین تولید کنیم. در این پژوهش، ما برای اولین بار ایده کلاس Mock نمادین را مطرح می‌کنیم بدین ترتیب که کلاس Mock نمادین کلاســی ا ست که تابع‌های کلاس ا صلی را دارد با این تفاوت که بدنه تابع حذف می شود 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 برنامک‌های دیگر حذف شود. چون کلاس‌های منبع و آسیب‌پذیر هر دو بخشی از SDK هستند، پس از این جهت به شکل Mock پیاده سازی می شوند. همچنین چون مقادیر متغیرها و خروجی تابع‌های آنها در تحلیل آلایش ا ستفاده می شوند، (یعنی داده‌هایی آلایش شده ه ستند) پس باید این مقادیر و خروجی‌ها را نمادین کنیم. نمونه کلاس Mock نماین برای تابع منبع در شکل ۶ آمده ا ست. بخشی از کلاس Mock نماین برای تابع آ سیب‌پذیر کلاس SQLiteDatabase در شکل ۱۰ دیده می شود. همان طور که دیده می شود تمام خروجی‌های تابع‌ها نمادین تولید می شوند. کد کامل این کلاس در پیوست آمده است.

```
public Cursor rawQueryWithFactory(CursorFactory cursorFactory, String sql, String[]
selectionArgs, String editTable,
CancellationSignal cancellationSignal) { return new Cursor() {
        @Override
        public void setExtras(Bundle extras) {          }
        @Override
        public Bundle respond(Bundle extras) {
                return (Bundle) Debug.makeSymbolicRef("dbCursor.Bundle", extras);
        }
        @Override
        public boolean requery() {
                return Debug.makeSymbolicBoolean("dbCursor.requery()");
        }
        @Override
        public boolean moveToPrevious() {
                return Debug.makeSymbolicBoolean("dbCursor.moveToPrevious()");
        }
        @Override
        public boolean moveToPosition(int position) {
                return     Debug.makeSymbolicBoolean("dbCursor.moveToPosition()");

        }
```

شکل ۱۰ تکه کدی از کلاس Mock نمادین تولید شده برای کلاس SQLiteDatabase

۱-۲- اجرای پویا-نمادین همراه با تحلیل آلایش توسط SPF اصلاح شده

برای تشـخیص آسـیب‌پذیری ورودی‌های برنامک (مثلا EditText) را نمادین در نظر گرفتیم. همچنین یک مولفه جدید به SPF اضـــافه کردیم. در این مولفه اجرای برنامه را ادامه می‌دهیم تا زمانی که به تابع آ سیب‌پذیر (مثلا query) بر سیم. سپس نمادین بودن ورودی تابع آ سیب‌پذری را برر سی می‌کنیم. در صورت نمادین بودن، پارامتری نبودن تابع آ سیب‌پذیر 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) بر سیم. در صورتی که ورودی تابع نشت، از Mock نمادین آسـیب‌پذیر باشـد، اطلاعات مورد نیاز برای تحلیل را اعلام می‌کنیم. این اطلاعات شـامل شناسه تابع منبع، شناسه تابع نشت، دنباله پشته برنامه تا تابع آسیب‌پذیر و تصفیه شدن یا نشدن داده

```
----------Vulnerability Detection Result----------
---STACK TRACE OF CURRENT APPLICATION RUN FOR CATCHING VULNERABILITY-------
1)
android.database.sqlite.SQLiteDatabase.rawQueryWithFactory(SQLiteDatabase$Curs
orFactory,String,String[],String,CancellationSignal)
2)android.database.sqlite.SQLiteDatabase.rawQuery(String,String[])
3) com.example.lab.testak_textinput.MainActivity$2.onClick(View)
4)com.example.lab.testak_textinput.dummyMain.main(String[])
--------END OF STACK TRACE----------
----------INFO OF INPUTS OF APP THAT CAUSE INJECTION VULNERABILIT----------
1)R.id.editText developer sanitizer for this input is OFF
------END OF NAME OF IDS----------
--------INFO OF OBJECT THAT CAUSE LEAKAGE----------
1)Method is android.widget.TextView.setText()
----------END OF OBJECT INFO----------
```

شکل ۱۱ نمونه خروجی ابزار برای تشخیص آسیب‌پذیری تزریق SQL

ورودی توسط برنامه‌نویس است. در شکل ۱۱ نمونه خروجی تولید شده برای یک برنامک آسیب‌پذیر آمده است.

۱-۲- 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 ۱۲ 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 شکل ۹ با ۲ شکل ۱۲، خط ۵ شکل ۹ با ۳ شکل ۱۲و ۶ شکل ۹ با ۷ شکل ۱۲یکی است). همچنین لازم است تا رشته‌هایی مثل «a’ or ‘1’=’1» را به ورودی آسیب‌پذیر برنامه بدهیم که معمولا موجب بهره‌جویی از آن می‌شود. (خط ۶ شکل ۱۲) برای بررسی بهره‌جویی، خروجی تابع نشت یعنی TextView را مشاهده کردیم (خط ۸ شکل ۱۲). برای یافتن شناسه تابع ورودی آسیب‌پذیر و شناسه تابع نشت از خروجی تحلیل پویا-نمادین استفاده کردیم.

```
1: public void SqlInjectionExploitability() 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 ۱۲ نمونه کد بهرجو در Robolectric

فصل پنجم

ارزیابی و جمع‌بندی

ارزیابی و جمع‌بندی

در این فصل به ارزیابی راه‌کار ارائه شده در فصل چهارم خواهیم پرداخت. در فصل چهارم از دو دیدگاه و با دو سوال پژوهشی در مورد آزمون برنامک‌های اندرویدی و تشخیص آسیب‌پذیری در این برنامک‌ها صحبت کردیم. در مورد آزمون برنامک‌های اندرویدی یک هیوریستیک ارائه کردیم که بر مبنای ترکیب تحلیل ایستا و پویا کار می‌کرد. ایده اصلی کار در این بود که در اجرای پویا-نمادین درخت اجرا به صورت پیش‌فرض به شکل عمق‌اول پیمایش می‌شود. در این پژوهش ما با استفاده از تحلیل ایستا و به کارگیری «گراف کنترل جریان بین تابعی[1]» مجموعه‌ای اطلاعات در مورد اولویت اجرای شاخه‌ها در دستورات شرطی جمع‌آوری می‌کنیم. سپس این اولویت را در زمان اجرای پویا-نمادین اعمال می‌کنیم. اعمال این اولویت‌ها باعث می‌شود، شاخه‌های دارای خطا سریع‌تر اجرا شوند.

در سوال پژوهشی دوم به دنبال تشخیص آسیب‌پذیری تزریق در برنامک‌های اندرویدی بودیم. برای جواب به این سوال تحلیل آلایش را به اجرای پویا-نمادین اضافه کردیم. عملا با در نظر گرفتن مقادیر ورودی به صورت نمادین (همان آلایش شده در تحلیل آلایش) و اجرای پویا-نمادین برنامک به این هدف ر سیدیم. در این کار از ایده Mock نمادین برای توابع آ سیب‌پذیر ا ستفاده کردیم. این ایده باعث شد که تحلیل ما کامل شود. در انتها با تکمیل موتور SPF و اضافه کردن مولفه تشخیص آسیب‌پذیری، ایده خود را تکمیل کردیم.

در این فصل هر یک از سوال‌های پژوهشی بالا را مورد آزمون و ارزیابی قرار داده‌ایم که به تفصیل به بیان هریک خواهیم پرداخت. در پایان به جمع‌بندی بحث و کارهای آینده خواهیم پرداخت.

[1] ICFG

۱-۲- ارزیابی هیوریستیک ارائه شده برای اجرای هدایت شده پویا-نمادین

ارزیابی راه‌کار ارائه شده در این بخش 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 پیاده سازی 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 [19] انتخاب شده‌اند. در جدول ۲ اطلاعات این برنامک‌ها آمده است که میزان پیچیدگی آنها را نشان می‌دهد.

ابزار Sig-Droid به منظور ارائه اجرای نمادین در برنامک‌های اندرویدی در سال ۲۰۱۵ ارائه شده است. در این ابزار با ایده تحلیل ایستا و گراف فراخوانی توابع نقاط شروع برنامه استخراج می شوند. سپس با استفاده از کلاس‌های Mock، چالش وابستگی به SDK حل شده است. در این کار بعد از فراهم آمدن برنامک Mock شده، از SPF به عنوان موتور اجرای نمادین استفاده می شود. نویسنده عنوان می‌کند با هدف پوشش کد هر چه بیشتر این ابزار پیاده سازی شده است. در نهایت این ابزار با ابزارهای مطرحی

چون Monkey و Dynodroid [20] مقایسه شده است و نشان داده شده است که این ابزار می‌تواند به پوشش بهتری از کد دست یابد.

در این سوال پژوهشی ما به دنبال این موضوع بوده‌ایم که با استفاده از تحلیل ایستا و به کار گیری گراف کنترل جریان بین تابعی مسیر اجرا را دقیق‌تر کنیم و با محدود کردن تحلیل به تابع‌های نقطه شروع مطلوب که ما را به خطا می‌رسانند، با پوشش کد کمتر و سرعت بیشتر بتوانیم خطا در برنامک را تشخیص دهیم.

جدول ۲ مشخصات برنامک‌های دنیای 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 ۳ اطلاعات مربوط به تحلیل برنامک‌های جدول ۲ با ابزار Sig-Droid و کار خودمان را آورده‌ایم. همان طور که دیده می‌شود ابزار ما در زمان کمتر با پوشش کمتر کد می‌تواند خطا را تشخیص دهد. دلیل این امر این است که ما با تحلیل ایستا و استفاده از گراف فراخوانی توابع، اجرا را محدود به تابع‌هایی می‌کنیم که در رسیدن به خطا نقش دارند. این موضوع پوشش کد را کاهش می‌دهد. 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.

نکته‌ای که باید به آن توجه کنیم این اســـت که در این تحلیل اگر ما اجرای پویا-نمادین را بدون اولویت‌گذاری شاخه‌ها اجرا کنیم و همچنین همه تابع‌های نقطه شروع را مورد آزمون قرار دهیم و علاوه‌بر آن به جای اجرای پویا-نمادین برنامه را به صـــورت نمادین اجرا کنیم، کار ما همان Sig-Droid خواهد بود و نتایج مشابه مقاله آن ابزار خواهد شد.

جدول ۳ مقایسه ابزار ما با Sig-Droid

ردیف	نام برنامک	Sig-Droid		کار ما	
		پوشش کد	زمان (ms)	پوشش کد	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

در جدول ۴، مقایســه ابزارهای مختلف با کار ما بر اســاس معیارهای موجود در مقالات [21] [14] [22] آمده است. در معیارهای مقایسه انتخاب شده به ترتیب روش جست‌وجو، انواع رخدادهای پشتیبانی شده در ابزار، ترکیب تحلیل ایســتا و پویا به عنوان هیوریســتیک در بهینه ســازی اجرا و چالش انفجار مســیر مطرح شــده‌اند. همان طور که پیش از این گفته شــد برای آزمون برنامک‌های اندرویدی ۳ روش متداول مطرح است. این روش‌ها عبارتند از :آزمون بر اساس ورودی‌های دلخواه و بی‌قاعده، آزمون مبتنی بر مدل و آزمون نظام‌مند. اجرای نمادین و پویا-نمادین از جمله روش‌های نظام‌مند محسوب می‌شوند. رخدادهای موجود در سیستم عامل اندروید در سه دسته رخدادهای متنی، سیستمی (مانند دریافت پیامک) و واسط گرافیکی[۱] (مانند فشردن یک دکمه) قرار می‌گیرند. لازم به ذکر است که مسئله انفجار مسیر در ابزارهای

[۱] GUI

Monkey و Swifthand مطرح نمی شود چون این دو ابزار مسیرهای مختلف موجود در کد را بررسی و اجرا نمی‌کنند.

همان طور که در جدول هم دیده می‌شـــود، کار ما از روش پویا-نمادین به عنوان یک روش نظام‌مند بهره می‌برد و همان طور که پیش از این گفته شد، این روش به نسبت تولید ورودی دلخواه و همچنین اجرای نمادین می‌تواند به پوشـــش بهتری از کد برســـد. در کار ما با اســـتفاده از ایده Mock و Mock نمادین می‌توانیم انواع رخدادهای مطرح را پشتیبانی کنیم. با استفاده از هیوریستیک ارائه شده که ترکیب تحلیل ای ستا و پویا ه ست تواند ستیم با سرعت بی شتر و پو شش کد کمتر خطاها را پیدا کنیم، همین مو ضوع موجب می‌شود که با چالش انفجار مسیر روبه‌رو نشویم.

جدول ۴ 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	مبتنی بر مدل	متن،GUI	×	-
Sig-Droid	نمادین	متن،GUI	×	×
کار ما	پویا-نمادین	متن،سیستم،GUI	✓	✓

۱-۲- ارزیابی تشخیص آسیب‌پذیری تزریق در برنامک‌های اندرویدی

ارزیابی راه‌کار ارائه شـــده در این بخش را در دو مرحله انجام دادیم. ابتدا برای نشـــان دادن این که روش در ست کار می کند ۱۰ برنامک را خودمان پیاده سازی کردیم. ۶ برنامک را به منظور را ستی‌آزمایی تولید کلاس dummyMain و تابع نقطه شـــروع، همچنین درســـت بودن فرایند تولید کلاس‌های Mock و

درســـت بودن اجرای پویا-نمادین و ارتباط این مولفه‌ها با هم پیاده‌ســـازی کردیم. در این برنامک‌ها برای نشان دادن وجود خطا، استثنای زمان‌اجرا را در برنامک‌ها قرار دادیم. در این ۶ برنامک مرحله به مرحله و از ساده‌ترین حالت تا شکل‌های پیچده را پیاده سازی 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 آســـیب‌پذیر را در دو حالت اســـتفاده از حالت پارامتری و غیرپارامتری آنها پیاده‌ســـازی کنیم. همچنین حالت‌های مختلف جریان داده در برنامک (مثلا اســـتفاده از Intent و یا اســـتفاده از پایگاه‌داده برنامک دیگر) را پیاده سازی کردیم. با استفاده از این برنامک‌ها مولفه جدید پیاده سازی شده در SPF را آزمودیم. این مولفه را برای تشخیص آسیب‌پذیری تزریق SQL پیاده‌سازی کرده بودیم.

برای اینکه نشـــان دهیم روش ما در برابر برنامک‌های واقعی هم درســـت کار می‌کند، از مخزن F-Droid نیز اســـتفاده کردیم. این مخزن شـــامل برنامک‌های اندرویدی متن‌باز در موضـــوعات مختلف اســـت. ۱۴۰ برنامک مختلف را به دلخواه انتخاب کردیم. برای تحلیل ابتدا سعی کردیم برای برنامک‌ها تابع نقطه شروع بســـازیم. از این برنامک‌ها، برای ۷ برنامک، تابع نقطه شـــروع تولید شـــد. این نشـــان 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 شـــده آســـیب‌پذیر به تزریق SQL بودند که تنها در یک مورد برنامه‌نویس از تابع پارامتری استفاده کرده 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.

در جدول ۵ مقایسه ابزار ما با ابزارهای مشابه فعلی از ۵ جنبه مطرح در مقالات آمده است[14][15][13]. ابزارهای Condroid و APPINTENT که در این جدول با کار ما مقایسه شده‌اند دغدغه امنیتی دارند. Condroid با اســـتفاده از اجرای پویا-نمادین به دنبال کشـــف بمب منطقی در برنامک‌های اندرویدی اســـت. APPINTENT با اجرای نمادین به دنبال کشـــف نقض حریم خصـــوصـــی توســـط برنامک‌های اندرویدی است. از این جهت با کار ما که به دنبال تشخیص آسیب‌پذیری تزریق در برنامک‌ها است قرابت دارند. ابزار Sig-Droid هم همان طور که گفته شـــد، به منظور آزمون برنامک‌ها و رســـیدن به پوشـــش بالای کد با استفاده از اجرای نمادین است. ولی از آنجایی که ایده‌های مطرح در این کار با کار ما شباهت دارد، در این جدول آمده است.

معیارهایی که در این جدول آمده‌اند، معیارهای مهم و مطرح در مقایسه ابزارهای تحلیل امنیتی در حوزه اندروید هستند. ابزارهایی که در حوزه امنیتی ارائه می شوند در سه دسته تشخیص دژافزار، تشخیص آسیب‌پذیری و تشخیص نقض حریم خصوصی قرار می‌گیرند. همان طور که دیده می‌شود کارهای کنونی همگی مسئله رخدادمحور بورن را به گونه‌ای حل کرده اند. ابزار APPINTENT با استفاده از تحلیل ایستای آلایش سعی کرده است تا اجرای نمادین خود را به صورت هدایت شده انجام دهد این موضوع باعث شده است تا مسئله انفجار مسیر را حل کند.

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.

جدول ۵ مقایسه ابزار ارائه شده با ابزارهای مشابه موجود

معیار / مقایسه / ابزار	رخدادمحور بودن	عدم انفجار مسیر	ترکیب تحلیل ایستا و پویا	تشخیص بمب منطقی	تشخیص آسیب‌پذیری	تشخیص نقض حریم خصوصی
APPINTENT	✓	✓	✓	×	×	✓
Condroid	✓	×	×	✓	×	×
Sig-Droid	✓	×	×	×	×	×
کار ما	✓	✓	✓	×	✓	×

۱-۲- جمع‌بندی و کارهای آینده

منابع و مراجع

[1] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 213–223, 2005.

[2] K. Sen, D. Marinov, G. Agha, K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," *10th European software engineering conference and 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE'05)*, vol. 30, no. 5, p. 263, 2005.

[3] K. Sen and G. Agha, "A race-detection and flipping algorithm for automated testing of multi-threaded programs," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4383 LNCS, pp. 166–182, 2007.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe," *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06*, vol. 12, no. 2, p. 322, 2006.

[5] R. Majumdar and K. Sen, "Hybrid concolic testing," *Proceedings - International Conference on Software Engineering*, pp. 416–425, 2007.

[6] P. Godefroid, "Compositional Dynamic Test Generation," *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 47–54, 2007.

[7] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 209–224, 2008.

[8] K. Kähkönen, T. Launiainen, O. Saarikivi, J. Kauttio, K. Heljanko, and I. Niemelä, "LCT: An open source concolic testing tool for Java programs," *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, pp. 75–80, 2011.

[9] T. Avgerinos, S. Cha, B. Hao, and D. Brumley, "AEG: Automatic Exploit Generation.," *Ndss*, 2011.

[10] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," *Proceedings of the ACM SIGSOFT 20th International*

Symposium on the Foundations of Software Engineering - FSE '12, p. 1, 2012.

[11] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," *Proceedings - IEEE Symposium on Security and Privacy*, pp. 380–394, 2012.

[12] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: a selective record-replay and dynamic analysis framework for JavaScript," *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, p. 488, 2013.

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

[14] 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.

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

[16] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," *Proceedings 2016 Network and Distributed System Security Symposium*, no. February, pp. 21–24, 2016.

[17] P. A. S. and R. S. and F. C. and B. E. and B. A. and K. J. and L. T. Y. and O. 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.

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

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

[20] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for Android apps," *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, p. 224, 2013.

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

[22] H. A. S. and N. M. and H. Mary Jean and Yang, "Automated concolic testing of smartphone apps," in *FSE '12: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, p. 59.

.

پیوست‌ها

موضوعات مرتبط با متن گزارش پایان نامه که در یکی از گروه‌های زیر قرار می‌گیرد، در بخش پیوست‌ها آورده شوند:

۱- اثبات های ریاضی یا عملیات ریاضی طولانی.

۲- داده و اطلاعات نمونه (های) مورد مطالعه (Case Study) چنانچه طولانی باشد.

۳- نتایج کارهای دیگران چنانچه نیاز به تفصیل باشد.

۴- مجموعه تعاریف متغیرها و پارامترها، چنانچه طولانی بوده و در متن به انجام نرسیده باشد.

برای شماره‌گذاری روابط، جداول و اشکال موجود در پیوست از ساختار متفاوتی نسبت به متن اصلی استفاده می‌شود که در زیر به‌عنوان نمونه نمایش داده شده‌است.

$$F = ma \qquad (پ-۱)$$

جدول پ-۱: شرح کد منبع بدنه اصلی یک کد رایانه‌ای.

```
01 program AeroPack;
02 uses
03   Forms,
04   Unit1 in 'Unit1.pas' {Form1},
05   Dialogs,
06   Sysutils;
07 {$R *.res}
08 begin
09   Application.Initialize;
10   Application.Title := 'AeroPack';
11   Application.CreateForm(TForm1, Form1);
12   if pos('/h',Form1.Switches)<>0 then
13   begin
14    Application.ShowMainForm:=False;
15    Form1.Visible:=False;
16   end;
```

در صورتیکه سوئیچ h/ در رشته سوئیچ موجود باشد، متغیر ShowMainForm و خصوصیت Visible فرم اصلی را برابر با False قرار می‌دهد. نتیجه این کار عدم نمایش فرم اصلی خواهد بود.

```
17   Application.Run;
18 end.
```

Abstract

This page is accurate translation from Persian abstract into English.

Key Words: Write a *3* to *5* KeyWords is essential.

Amirkabir University of Technology
(Tehran Polytechnic)

… Department …

MSc or PhD Thesis

Title of Thesis

By
Name

Supervisor
Dr.

Advisor
Dr.

Month & Year