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

با سپاس از سه وجود مقدس:

آنان که ناتوان شدند تا ما به توانایی برسیم...

موهایشان سپید شد تا ما روسفید شویم...

و عاشقانه سوختند تا گرمابخش وجود ما و روشنگر راهمان باشند...

پدرانمان

مادرانمان

استادانمان

تقدیر و تشکر:

سپاس و ستایش مر خدای را جل و جلاله که آثار قدرت او بر چهره روز روشن، تابان است و انوار حکمت او در دل شب تار، درفشان. آفریدگاری که خویشتن را به ما شناساند و درهای علم را بر ما گشود و عمری و فرصتی عطا فرمود تا بدان، بنده ضعیف خویش را در طریق علم و معرفت بیازماید.

بدون شک جایگاه و منزلت معلم، بالاتر از آن است که در مقام قدردانی از زحمات بی شائبه‌ی او، با زبان قاصر و دست ناتوان، چیزی بنگارم. اما از آنجا که تجلیل از معلم، سپاس از انسانی است که هدف آفرینش را تامین می‌کند، به رسم ادب دست به قلم برده‌ام، باشد که این خردترین بخشی از زحمات آنان را سپاس گوید.

از پدر و مادر مهربانم، این دو معلم بزرگوار که همواره بر کوتاهی من، قلم عفو کشیده و کریمانه از کنار غفلت‌های گذشته‌اند و در تمام عرصه‌های زندگی یار و یاورم بوده‌اند؛

از استاد با کمالات، جناب آقای دکتر بابک صادقیان که در کمال سعه صدر، با حسن خلق و فروتنی، از هیچ کمکی در این عرصه بر من دریغ نداشتند؛

از اساتید محترم، جناب آقای دکتر ... و آقای .... که زحمت داوری این رساله را متقبل شدند؛

و در پایان، از حمایت‌ها و دلسوزی‌های دوستان عزیزم، آقایان سید محمد مهدی احمدپناه، محمود اقوامی پناه، سید امیرحسین ناصرالدینی، حمیدرضا رمضانی و احمد اسدی که در طول پروژه از راهنمایی‌هایشان استفاده کردم؛

کمال تشکر و قدردانی را دارم.

چکیده

برنامک‌های گوشی‌های هوشمند از جمله نرم‌افزارهای محبوب میان مردم هستند. استفاده از این برنامک‌ها روز به روز افزایش می‌یابد. از میان سیستم عامل‌های مطرح در گوشی‌های هوشمند، اندروید محبوبیت زیادی دارد. علاوه بر آن متن‌باز بوده و می‌توان به کدها و کتابخانه‌های چارچوبه‌کاری آن دسترسی داشت. محبوبیت این برنامک‌ها اهمیت آزمون نرم‌افزار و آزمون امنیتی آنها را بالا می‌برد. از اجرای پویا-نمادین برای آزمون نرم‌افزارهای مختلف استفاده می‌شود. این اجرا دارای پوشش قوی کد است و می‌تواند خطاها و آسیب‌پذیری‌های موجود در آن را دقیق‌تر و بدون مثبت نادرست کشف کند. آزمون برنامک‌های اندرویدی نسبت به برنامه‌های دیگر دارای چالش‌های جدید رخدادمحوربودن و وابستگی زیاد به SDK است که سربار آزمون را بالا می‌برد. در این پژوهش ما به دو سوال پژوهشی پاسخ داده‌ایم که در ادامه توضیح داده می‌شوند.

در این پژوهش روشی ارائه می‌شود که با اجرای پویا-نمادین همراه تحلیل آلایش به دنبال تشخیص آسیب‌پذیری تزریق در برنامک‌های اندرویدی هستیم. در این کار با تحلیل ایستا، گراف فراخوانی توابع و پیمایش برعکس از تابع آسیب‌پذیر تا تابع منبع، نقطه شروع برنامک 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 در سوال پژوهشی اول، 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.

همچنین، در این پژوهش ما یک هیوریستیک را ارائه کرده‌ایم که اجرای پویا-نمادین را به صورت بهینه و هدایت شده روی برنامک‌های اندرویدی اعمال می‌کند. با استفاده از گراف کنترل جریان بین تابعی و پیمایش روبه‌عقب آن، اطلاعات مسیرهای دارای اولویت را در یک پشته ذخیره می‌نماییم. 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 در سوال پژوهشی دوم، 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 پوشش کد کمتر و با سرعت بیشتری می‌تواند خطاهای برنامک را تشخیص دهد.

واژه‌های کلیدی:

اجرای پویا-نمادین، اجرای پویا-نمادین هدایت شده، تشخیص آسیب‌پذیری، آسیب‌پذیری تزریق، برنامک‌های اندرویدی

فهرست مطالب

صفحه

فهرست اشکال

صفحه

فهرست جداول

صفحه

۱

فصل اول

مقدمه

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. اندروید محبوب‌ترین سیستم عامل حال حاضر گوشی‌های هوشمند است. با گسترش اندروید، توسعه برنامک‌های اندرویدی نیز رشد چشمگیری داشته اند 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.[1] 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[1] 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 آزمون و تشخیص آسیب‌پذری را با چالش روبه‌رو می‌کند. به طور معمول برنامک‌های اندرویدی با زبان جاوا پیاده‌سازی می‌شوند. این موضوع معمولا باعث به وجود آمدن این تصور می‌شود که برنامک‌های اندرویدی با برنامه‌های به زبان جاوا که برای پلتفرم‌هایی مثل کامپیوتر شخصی پیاده‌سازی می‌شوند، تفاوتی ندارند. اما این تصور یک تصور ساده انگارانه است. در زیر تفاوت‌های عمده اجرای نمادین و پویا-نمادین برنامک‌های اندرویدی و برنامه‌های به زبان جاوا عنوان می‌شوند. لازم به ذکر است که تا به حال موتور اجرای پویا-نمادین و یا موتور اجرای نمادین برای برنامک‌های اندرویدی پیاده‌سازی نشده است. دلیل این موضوع هم چالش‌های موجود در آزمون این برنامک‌ها است. در این پژوهش ما از موتورهای اجرای پویا-نمادین برنامه‌های به

[1] Software Development Kit

زبان جاوا کمک خواهیم گرفت که نزدیک‌ترین ابزار برای کار ما خواهند بود. ولی استفاده از آنها خود چالش‌هایی دارد که در ادامه توضیح خواهیم داد.

تفاوت آزمون برنامک‌های اندرویدی با سایر برنامه‌ها را می‌توان در سه مورد عنوان کرد:

۱. برنامک‌های اندروید وابسته به مجموعه ای از کتابخانه‌هایی هستند که در بیرون از دستگاه یا شبیه‌ساز[۲] در دسترس نیستند. کد اندروید در DVM[۳] اجرا می‌شود. برخلاف کدهای برنامه‌های جاوا که در JVM[۴] اجرا می‌شوند. پس به جای بایت‌کد جاوا برنامک‌های اندرویدی به بایت‌کد Dalvik کامپایل می‌شوند. برنامه‌های به زبان جاوا همگی دارای «تابع نقطه شروع[۵]» به برنامه هستند. یعنی این برنامه‌ها بدون استثنا از این تابع شروع به اجرا می‌شوند. ولی برنامک‌های اندرویدی چنین تابعی ندارند و اجرای یک برنامک به شکل‌های مختلفی امکانِ اجرا دارد. برای مثال ممکن است یک برنامک با باز کردن مستقیم آن شروع شود. در حالتی دیگر ممکن است این برنامک با اتفاق افتادن یک رخداد به‌خصوص، مثلا دریافت یک پیامک، شروع به اجرا کند. این موضوع خود چالشی جدی در تحلیل و آزمون این برنامک‌ها با موتور اجرای پویا-نمادین برای برنامه‌های به زبان جاوا خواهد بود. همچنین این برنامک‌ها باید به نحوی تغییر پیدا کنند که بتوان آنها را در JVM کامپایل کرد.

۲. برنامک‌های اندرویدی بسیار وابسته به کتابخانه‌های چارچوبه کاری یعنی SDK هستند و این موضوع باعث ایجاد مشکل واگرایی مسیر[۶] [2] می‌شود. در اجرای نمادین اگر یک مقدار نمادین از زمینه[۷] برنامه خارج شود، مثلا برای انجام یک پردازش به یک کتابخانه داده شود یا در اختیار چارچوبه کاری قرار گیرد، گفته می‌شود که واگرایی مسیر اتفاق افتاده است. واگرایی مسیر موجب ایجاد دو مشکل می‌شود:

- موتور اجرای نمادین ممکن است نتواند کتابخانه خارجی را اجرا کند پس تلاش بیشتری لازم است تا بتوان آن کتابخانه را نیز به صورت نمادین اجرا کرد.

---

[2] Emulator

[3] Dalvik Virtual Machine

[4] Java Virtual Machine

[5] Main Method

[6] Path divergence

[7] Context

- در کتابخانه خارجی ممکن است تعدادی قید وجود داشته باشد که در خروجی و حاصل پردازش کتابخانه موثر باشند. از این جهت این قیدها در تولید موردآزمون‌ها موثر خواهند بود و به جای آزمون برنامک اصلی، تمرکز به آزمون مسیر واگرا شده در کتابخانه معطوف می‌شود و به طور پیوسته لازم است تا قسمتی از سیستم عامل اندروید به صورت نمادین اجرا شود که در کل موجب ایجاد سربار زیاد در آزمون برنامک می‌شود.

یک مثال پرکاربرد از این نوع می‌تواند Intentها باشد که سیستم پیام‌رسانی بین مولفه[8]های مختلف در اندروید است. به وسیله Intent یک مقدار به یک مولفه در درون یک برنامه یا به مولفه‌ای در برنامه دیگر ارسال می‌شود. Intent بعد از خارج شدن از محدوده برنامه وارد کتابخانه‌های سیستمی‌شده و بعد از آن وارد مولفه مقصد می‌شود.

۳. برنامه‌های اندروید رخدادمحور[9] هستند. به این معنی که در اجرای نمادین، موتور اجرا باید منتظر کاربر بماند تا با تعامل با برنامک یک رخداد مثل لمس صفحه نمایش ایجاد شود. علاوه بر کاربر برنامه‌های دیگر هم می‌توانند رخداد تولید کنند مثل رخداد تماس ورودی یا دریافت یک پیام.

پیش از این پژوهش کارهایی در حوزه آزمون برنامک‌های اندرویدی با سه رویکرد متفاوت انجام شده است. در رویکرد اول به طور بی‌قاعده[10] ورودی برنامک تولید می‌گردد. در ابزار Monkey [3] به صورت دلخواه سعی می‌شود تا ورودی‌های 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[11] تولید می‌گردد. در ابزار Sig-Droid [4] با استفاده از یک روش مشخص مانند اجرای نمادین به صورت جعبه‌سفید[12] 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

[8] Component

[9] Event Driven

[10] Random

[11] Systematic

[12] White Box

پوشش هرچه بیشتر این مسیرها است. در رویکرد سوم مانند ابزار Swifthand [5] مدلی از برنامه مانند مدل رابط گرافیکی کاربر[۱۳] از برنامک استنتاج می‌گردد و سپس بر اساس این مدل‌ها ورودی‌هایی برای برنامک تولید می‌شود که مسیرهای ناشناخته برنامک را بپیماید.

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 [6] ارائه شد. 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[۱۴] وجود نداشته باشد و همچنین با روش‌های ضد ایستا مقابله کرد. چالش‌هایی که در این روش وجود دارد عبارتند از:

- انفجار مسیر: در برنامک‌های دنیای واقعی تعداد خطوط برنامک بسیار زیاد هستند. این موضوع باعث می‌شود مسیرهایی از برنامه که باید مورد تحلیل قرار گیرند به صورت نمایی افزایش پیدا کنند.
- چارچوب کاری و مدل‌سازی محیط: برنامک‌های اندرویدی در یک چارچوبه‌کاری خاص به خود اجرا و تولید می‌شوند. همان طور که پیش از این نیز گفته شد، برنامک‌های اندرویدی فراخوانی‌های زیادی به SDK دارند. همچنین این برنامک‌ها رخدادمحور هستند. در اجرای پویا-نمادین باید رخدادهای مختلف از جمله رخدادهای مرتبط با تعامل کاربر با برنامک نیز مدل‌سازی شوند تا فرایند تحلیل و آزمون شبیه به اجرای واقعی برنامک باشد.

در ادامه کارهایی را شرح خواهیم داد که در حوزه آزمون ویا آزمون امنیتی برنامک‌های اندرویدی با اجرای پویا-نمادین صورت گرفته است. برای اولین بار ACTEVE [7] 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[۱۵] 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

[13] GUI

[14] False Posetive

[15] Tap Event

با انفجار مسیر روبه‌رو شود. در [8] از ابزار پیشین با عنوان Condroid برای تشخیص وجود دژافزار[16]ها استفاده شده است که همان اشکالات ACTEVE را به ارث برده است. در AppIntent [9] از اجرای پویا-نمادین برای کشف نقض حریم خصوصی در برنامک‌های اندرویدی استفاده شده است. این ابزار به تحلیل‌گر انسانی کمک می‌کند تا فرایند کشف نشت اطلاعات حساس سریع‌تر اتفاق بیفتد. ابزار بعدی Sig-Droid است که 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 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.

بر اساس آخرین دانسته ما تاکنون [10]، ابزاری برای تشخیص آسیب‌پذیری در برنامک‌های اندرویدی با اجرای پویا-نمادین ارائه نشده است. کارهایی که تا به حال در مورد تشخیص آسیب‌پذیری در برنامک‌های اندرویدی انجام شده است، همگی از تحلیل ایستای کد برنامک استفاده می‌کنند. در جایی هم اگر نامی از تحلیل پویا آمده است منظور اجرای پویا-نمادین و یا نمادین برنامک نیست. بلکه منظور اجرای برنامک در محیط شبیه‌ساز است که به این وسیله درستی تشخیص خود را مورد آزمایش قرار می‌دهند. از جمله این ابزارها و پژوهش‌ها می‌توان به TaintDroid [11]، ScanDroid [12]، CHEX [13]، APSET [14]، VulHunter [15] و کارهای [16] و [17] اشاره کرد.

در منبع [18] عنوان شده است که آسیب‌پذیری تزریق می‌تواند موجب نشت اطلاعات حساس کاربر شود. آسیب‌پذیری تزریق در اندروید می‌تواند تزریق دستور به پوسته[17] 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 داده‌ایم.

در این پژوهش ما دو سوال پژوهشی مطرح کرده‌ایم و به آنها پاسخ داده‌ایم:

[16] Malware

[17] Shell

۱. بهبود اجرای پویا-نمادین برنامک‌های اندرویدی برای تولید خودکار ورودی آزمون
۲. اجرای پویا-نمادین برای تشخیص آسیب‌پذیری تزریق در برنامک‌های اندرویدی

برای پاسخ به سوال اول، ابتدا ما apk برنامک را دیکامپایل می‌کنیم. سپس برای اینکه بتوانیم بر روی JVM برنامک را با موتور SPF[۱۸] [19] به شکل پویا-نمادین اجرا کنیم، لازم است تا تابع نقطه شروع به برنامک را تولید نماییم. برای این منظور با تحلیل ایستا و استخراج «گراف فراخوانی توابع[۱۹]» 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 رو به عقب «گراف کنترل جریان بین تابعی[۲۰]» بهره می‌بریم. در پیمایش رو به عقب، 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، برنامه 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[۲۱] خود را ارائه می‌دهیم که باعث می‌شود، به جای پیمایش عمق‌اول[۲۲] 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

[18] Symbolic Path Finder

[19] Call Graph

[20] Inter-Control Flow Graph (ICFG)

[21] Heuristic

[22] DFS

Robolectric [20] و ورودی‌های آزمون استخراج شده از اجرای پویا-نمادین، برنامک را اجرا می‌نماییم تا صحت عملکرد هیوریستیک و ورودی‌های تولید شده را بررسی کنیم.

برای ارزیابی راه‌کار ارائه شده، علاوه بر ۱۰ برنامکی که خودمان مطرح و پیاده‌سازی کردیم، ۴ برنامک از برنامک‌های مورد آزمون در ابزار Sig-Droid را آزمودیم. این برنامک‌ها از مخزن F-Droid [21] انتخاب شده‌اند که مخزن متن‌باز[۲۳] برنامک‌های اندرویدی هستند. نتایج نشان می‌دهد که هیوریستیک ما باعث می‌شود با سرعت بیشتر و پوشش کد کمتر به نسبت ابزار Sig-Droid به خطاهای موجود در برنامه برسیم.

برای پاسخ به سوال دوم، باز از موتور SPF استفاده کرده‌ایم. 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 آسیب‌پذیر[۲۴] (مثلا query) تا تابع منبع[۲۵] (مثلا EditText) به صورت برعکس پیمایش کردیم و تابع نقطه شروع 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. 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[۲۶] را به اجرای پویا-نمادین اضافه کردیم. برای اینکه بتوانیم گردش داده آلایش‌شده[۲۷] 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

[23] Open Source

[24] Sink Method

[25] Source Method

[26] Taint Analysis

[27] Tainted Data

دنباله پشته[۲۸] برنامه تا تابع آسیب‌پذیر، شناسه تابع منبع، تصفیه‌شدن[۲۹] یا نشدن داده ورودی توسط برنامه‌نویس و شناسه تابع نشت[۳۰] داده‌ها (مثلا TextView) است. در نهایت با استفاده از این خروجی‌ها و تابع ورودی بدست آمده از تحلیل ایستا با ابزار Robolectric قابلیت بهره‌جویی[۳۱] برنامک به تزریق SQL را بررسی کردیم.

برای ارزیابی راه‌کار ارائه شده در مجموع ۱۵۰ برنامک را مورد تحلیل قرار داده‌ایم. ۱۰ برنامک را خودمان مطرح و پیاده‌سازی کرده‌ایم و ۱۴۰ برنامک باقی مانده از مخزن F-Droid انتخاب شده‌اند. از این تعداد برنامک ۱۱ برنامک آسیب‌پذ به تزریق SQL بودند که ما همه را توانستیم تشخیص دهیم.

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

- اجرای برنامک روی JVM با استفاده از نقطه ورودی به دست آمده 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 تا اجرای شاخه‌های دارای خطا را اولویت دهیم.
- اضافه کردن تحلیل آلایش به اجرای پویا-نمادین به منظور تشخیص آسیب‌پذیری تزریق و اضافه کردن یک مولفه به SPF برای این منظور.
- محدود کردن فرایند تحلیل به تابع‌های نقطه شروعی که منجر به بروز آسیب‌پذیری می‌شوند و محدودکردن آن به کلاس‌ها و توابع مطلوب، با استفاده از گراف فراخوانی توابع و پیمایش روبه‌عقب در آن و تولید کلاس Mock برای بقیه کلاس‌ها و توابع موجود در برنامک.

---

[28] Stack Trace

[29] Sanitize

[30] Leackege Method

[31] Exploitability

- استفاده از ایده Mock نمادین برای کلاس و تابع‌های آسیب‌پذیر به منظور گردش داده‌های آلایش‌شده در برنامک و دنبال کردن آنها تا تابع نشت.

در ادامه نحوه سازمان‌دهی پایان‌نامه آمده است:

در فصل دوم، اجرای پویا-نمادین همراه با مثال توضیح داده خواهد شد. همچنین انواع این اجرا و چالش‌های مطرح در این حوزه بیان می‌شوند. در پایان این فصل کارهای مطرح صورت گرفته در این حوزه از سال ۲۰۰۵ تا کنون مورد بررسی قرار می‌گیرند و ویژگی‌ها و تفاوت‌های آنها با هم مقایسه خواهد شد.

در فصل سوم، مطالعه‌ای بر روی آسیب‌پذیری در نرم‌افزارها صورت گرفته است. در این فصل به طور کلی روش‌های تشخیص آسیب‌پذیری در نرم‌افزارها عنوان می‌شود. به عنوان نمونه آسیب‌پذیری تزریق به تفصیل مورد بررسی قرار خواهد گرفت. همچنین آسیب‌پذیری‌های مطرح در برنامک‌های اندرویدی، کارها و ابزارهای ارائه شده برای تشخیص آنها را معرفی خواهیم کرد. در پایان فصل هم آسیب‌پذیری تزریق SQL در برنامک‌های اندرویدی همراه با مثال بررسی خواهد شد.

در فصل چهارم، راه‌کار پیشنهادی خودمان را بیان می‌کنیم. در این فصل دو سوال پژوهشی مطرح شده را به تفصیل توضیح خواهیم داد و راه‌کار ارائه شده خودمان برای پاسخ گویی به این سوالات را همراه با مثال توضیح می‌دهیم.

در فصل پنجم، به ارزیابی راه‌کار پیشنهادی می‌پردازیم. در این فصل به صورت جداگانه راه‌کار ارائه شده برای هر یک از سوالات پژوهشی مطرح شده را مورد ارزیابی قرار می‌دهیم و ابزار خود را با ابزارهای مطرح در این حوزه مقایسه خواهیم کرد. در پایان این فصل به جمع‌بندی بحث و کارهای آینده مطرح در این حوزه می‌پردازیم. در فصل ششم و هفتم، منابع و مراجع و در نهایت پیوست‌ها خواهد آمد.

۲

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.

۲-۱ 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.

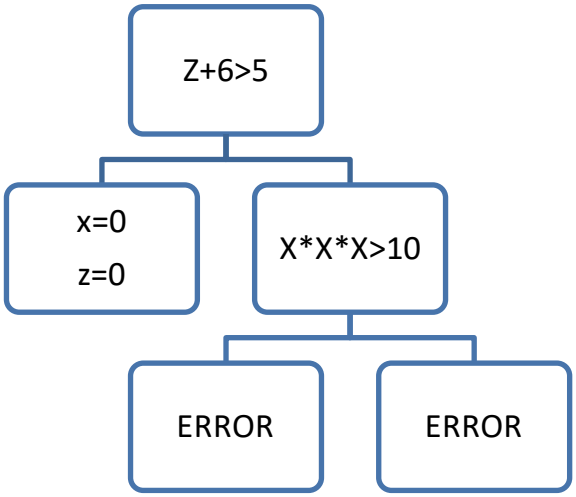
```
1:  testMe(int x, int y){
2:        if(y>5){
3:                assert(false);
4:        }else{
5:                if(x*x*x > 10){
6:                        assert(false);
7:                }
8:        }
9:  }
10: void main ( ){
11:       int x = symbolicinput();
12:       int z = symbolicinput();
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 می‌شود (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[۳۲] نگهداری می‌کند. همان طور که مشخص است دستورات شرطی موجود در برنامه موجب می‌شوند تا مجموعه دستورهایی که قرار است بعد از آن اجرا شوند، تصمیم‌گیری شوند. شرط مسیر عبارتی است که از عطف[۳۳] شرط‌های دستورهای شرطی موجود در آن مسیر بدست می‌آید. شرط مسیر در ابتدا مقدار درست[۳۴] دارد. برای مثال در خط ۲ برنامه، شرط y > 5 که معادل Z+6 > 5 است به شرط مسیر اضافه می‌شود و داریم: PC=(Z+6>5). اگر این شرط برقرار باشد، 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[۳۵]ی برنامه را می‌سازند. 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[۳۶]» داده می‌شود. این ابزارها با دریافت یک عبارت منطقی، مقادیر متناظر با هر متغیر در آن عبارت را پیدا می‌کنند که به ازای آنها کل عبارت درست خواهد بود. مثلا به ازای x=0 و y=0 عبارت

[32] Path Condition

[33] And

[34] True

[35] Execution Tree

[36] Constraint Solver

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[۳۷] نیز در نظر گرفته می‌شوند. مقدار عینی در ابتدا به صورت دلخواه انتخاب می‌شود. 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 می‌شود. 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.

۲-۲ 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 موجود در آن را اجرا می‌کند. همچنین سیستم فایل، شبکه، تعاملات کاربر با برنامه و

[37] Concrete

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.

۲-۳ 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 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

است، برنامه به صورت آنلاین اجرا می‌شود. بعد از آن اجرا آفلاین می‌شود تا زمانی که به اندازه کافی حافظه آزاد شود تا دوباره اجرا به شکل آنلاین ادامه پیدا کند.

۲-۴ کارهای گذشته

در ادامه کارهایی را بیان خواهیم کرد که در مورد اجرای پویا-نمادین از سال ۲۰۰۵ تا کنون انجام شده است. در مورد هر کار ویژگی‌های خاص آن و بهبودی که در مورد هر چالش داشته است را توضیح خواهیم داد. در جدول ۲-۱ کارهای گذشته شاخص آمده‌اند.

جدول ۲-۱ 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	۲۰۰۸	KLEE	C	کامپیوتر شخصی	آنلاین	EXE، بهینه‌سازی انتخاب مسیر و حل‌کننده قید، متن‌باز
۸	۲۰۰۹	jFUZZ	جاوا	کامپیوتر شخصی	آفلاین	متن‌باز، بهینه‌سازی کارهای گذشته
۹	۲۰۱۰	SPF	جاوا	کامپیوتر شخصی	آفلاین	ترکیب وارسی مدل و اجرای نمادین
۱۰	۲۰۱۱	LCT	جاوا	کامپیوتر شخصی	آفلاین	متن‌باز، SMT، معماری برنامه نویسی سوکت
۱۱	۲۰۱۱	AEG	باینری و C	کامپیوتر شخصی	آفلاین	آسیب‌پذیری سرریز بافر، خاصیت ایمنی
۱۲	۲۰۱۲	ACTEVE	اندروید	Phone	آفلاین	متن‌باز، رخدادمحور
۱۳	۲۰۱۲	MAYHEM	باینری	کامپیوتر شخصی	ترکیبی	آسیب‌پذیری سرریز بافر، قالب رشته، مدل‌سازی‌حافظه
۱۴	۲۰۱۳	Jalangi	جاوا اسکریپت	وب	آفلاین	ثبت-بازاجرای انتخابی، مقادیر سایه
۱۵	۲۰۱۳	AppIntent	اندروید	Phone	آفلاین	کشف نشت حریم خصوصی
۱۶	۲۰۱۵	SIG-Droid	اندروید	Phone	آفلاین	استفاده از کلاس‌های Mock، گراف فراخوانی توابع
۱۷	۲۰۱۵	Condroid	اندروید	Phone	آفلاین	پویا-نمادین Call Flow + Graph
۱۸	۲۰۱۶	Driller	باینری	کامپیوتر شخصی	ترکیبی	ترکیب instrumented-فازر با پویا-نمادین Genetic-

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 [۶] ارائه شد. این ابزار از lp solve به عنوان حل‌کننده قید استفاده می‌کند. همچنین محدود به زبان C است و مدل‌سازی حافظه ندارد. علاوه بر آن از برنامه‌های هم‌روندی پشتیبانی نمی‌کند. از جست‌وجوی DFS برای انتخاب مسیرها در درخت اجرا استفاده می‌کند و بهینه‌سازی 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 [۲۲] 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 مدل‌سازی 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: بررسی سریع ارضاناپذیری، حذف قیدهای معمول و حل افزایشی.

۳. در سال ۲۰۰۶، ابزار JCUTE [۲۳] با روش پویا-نمادین آفلاین، ارائه شد که از lpsolve استفاده می‌کند. این ابزار محدود به زبان جاوا است ولی مدل‌سازی 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 صورت خودکار برنامه‌ریزی شود. این ابزار از جست‌وجوی DFS برای انتخاب مسیرها استفاده می‌کند و مانند CUTE بهینه‌سازی برای ارسال قیدها به حل کننده قید دارد.

۴. در سال ۲۰۰۶، ابزار EXE [۲۴] با روش پویا-نمادین آنلاین، ارایه شد که از STP استفاده می‌کند. این ابزار محدود به زبان C است و از هم‌روندی پشتیبانی نمی‌کند. ولی مدل‌سازی حافظه دارد. حافظه را مجموعه‌ای از بایت‌های بدون نوع در نظر می‌گیرد. همچنین از جست‌وجوی DFS و BFS به صورت ترکیبی برای انتخاب مسیرها استفاده می‌کند. علاوه برآن بهینه‌سازی برای ارسال قیدها به حل کننده قید دارد. ایده‌های این ابزار در این مورد استفاده از روش کش و شناسایی زیرقیدهای مستقل و حذف زیرقیدهای بی ارتباط است.

۵. اجرای هیبرید [۲۵] به صورت ترکیبی اجرای دلخواه[۳۸] 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 می‌شود. هر گاه اجرا اشباع شد اجرا به پویا-نمادین تغییر میابد تا بتواند به صورت 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 مناسب است. این اجرا همان محدودیت‌های اجرای پویا-نمادین را دارد. ممکن است به پوشش ۱۰۰ درصد از کد نرسد ولی از نظر نویسندگان پوشش کامل نشانه ای برای قابل اعتماد بودن[۳۹] کد نیست.

۶. کار مورد شش در جدول [۲۶]، از DART به عنوان موتور پویا-نمادین استفاده می‌کند. هدف این کار توسعه DART برای برنامه های واقعی با تعداد خط کد بالاست به همین دلیل از تحلیل ایستای Compositional استفاده می‌کند که برای توابع function summery استخراج می‌کند و به جای اجرای هر باره یک تابع از summery آن استفاده می‌کند و آن را به شرط مسیر اضافه می‌کند.

۷. ابزار KLEE [۲۷] در سال ۲۰۰۸، با روش پویا-نمادین آنلاین ارائه شد که از STP استفاده می‌کند. این ابزار برای آزمون برنامه‌های واقعی محدود به زبان C است. مدل‌سازی محیط اجرای برنامه(سیستم فایل) و مدل‌سازی حافظه دارد. حافظه را مجموعه‌ای از بایت‌های بدون نوع در نظر می‌گیرد. ولی از هم‌روندی پشتیبانی نمی‌کند. این ابزار روش‌های انتخاب دلخواه و انتخاب برای پوشش بیشترین مسیرها را به صورت ترکیبی استفاده می‌کند. بر اساس یک سری هیوریستیک به حالت‌ها وزن اختصاص داده می‌شود و سپس به صورت دلخواه یکی از این حالت‌ها انتخاب می‌شوند. در حالت دوم، هیوریستیک‌ها بر اساس کمترین فاصله تا دستور پوشش داده نشده، بیشینه فراخوانی حالت و یا اینکه یک حالت اخیرا دستور جدیدی را پوشش داده است یا نه، محاسبه می‌شود. ترکیب این دو استراتژی باعث می‌شود هم پوشش تمامی دستورات فراهم شود و هم از گیر کردن در حلقه جلوگیری به عمل آید. برای بهینه‌سازی قیدها

[38] Random

[39] Reliability

به حل کننده قید از روش‌هایی مثل روش کش استفاده می‌کند. این ابزار گسترش یافته ابزار EXE است.

۸. jFuzz [28] ابزار متن باز برای جاواست. نوآوری خاصی ندارد و ترکیب بهینه سازی های کارهای قبلی مثل KLEE، CUTE و غیره را در خود دارد. این ابزار بروی پروژه JPF [29] پیاده سازی شده است.

۹. برای اجرای نمادین برنامه‌های به زبان جاوا، ابزار SPF [19] 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 پژوهش با تغییر SPF به دنبال تشخیص آسیب‌پذیری تزریق به برنامک‌های اندرویدی هستیم.

۱۰. ابزار LCT [۳۰] ابزار متن باز روی جاوا است. در این ابزار سعی شده از معماری کارگذار-کارخواه[40] 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 نیست. 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 غیره) در آن معنی ندارد. تنها با فریم های پشته و دستورات پرش و آدرس های حافظه سر و کار دارد. در AEG [۳۱] از ترکیب هر دو روش یعنی تحلیل باینری و کد برنامه استفاده شده است. نحوه کار AEG به این صورت است:

- ابتدا با استفاده از کد تحلیل نمادین صورت می‌گیرد تا به دستور آسیب‌پذیر برسد.
- سپس شرط مسیر به حل کننده قید داده می‌شود تا ورودی مناسب تولید شود.

[40] client-server

- سپس به صورت پویا و با استفاده از ورودی تولید شده، فایل باینری برنامه تحلیل می‌شود تا اطلاعات زمان‌اجرا[۴۱] یعنی ساختار حافظه مثل آدرس بافر سرریز شده و آدرس بازگشت استخراج شود.
- AEG قیدهای جدیدی مربوط به اطلاعات ساختار حافظه تولید می‌کند و به شرط مسیر اضافه می‌کند. این قیدها باید شامل shell code و آدرس بازگشت به shell code باشند. سپس شرط مسیر به حل کننده قید داده می‌شود تا ورودی مناسب تولید شود.
- در نهایت AEG ورودی تولید شده را به برنامه می‌دهد تا بررسی کند که کد بهرجو[۴۲] آیا اجرا می‌شود یا نه! اگر حل کننده قید نتواند شرط مسیر را حل کند، AEG آن را رها می‌کند و فرایند را ادامه می‌دهد.

۱۲. ابزار ACTEVE [۷] اجرای پویا-نمادین آفلاین برای برنامه‌های گوشی همراه است که از Z3 SMT solver استفاده می‌کند. این ابزار برای برنامک‌های اندرویدی[۴۳] ارائه شده است. این برنامک‌ها رخدادمحور[۴۴]، 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.

در این مقاله از روش پویا-نمادین برای تولید رخدادها استفاده می‌شود. برای این منظور SDK[۴۵] و برنامه تحت آزمون باید تجهیز[۴۶] شوند. سپس 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 بررسی شود. (دوتایی،

[41] Runtime

[42] Exploit

[43] Android Apps

[44] Event Driven

[45] Software Development Kit

[46] Instrument

سه‌تایی، چهارتایی و غیره) و جای‌گشت‌های مختلف رخدادها در هر ترتیب نیز در نظر گرفته شود که فضای حالت خیلی بزرگی دارد. این کار محدود به رخداد ضربه[۴۷] است. علاوه بر آن همان طور که گفته شد، نیاز به بهینه‌سازی برای کاهش فضای حالت در ترتیب‌های مختلف از رخدادها دارد. در این کار با حذف ویجت[۴۸]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 [۳۲] 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[۴۹] برنامه آسیب‌پذیر را دریافت می‌کند. به SES[۵۰] وصل می‌شود تا مقداردهی‌های اولیه صورت پذیرد. سپس کد به صورت عددی اجرا می‌شود و همزمان تحلیل آلایش[۵۱] پویا نیز روی آن اجرا می‌شود.
- 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:

[47] Tap Event

[48] Widget

[49] Concrete Execution Client

[50] Symbolic Execution Server

[51] Taint Analysis

- 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[۵۲] تولید و به SES داده می‌شود اگر قابل ارضا بود یعنی کد از این مسیر آسیب‌پذیر است.

۱۴. ابزار Jalangi [۳۳] 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[۵۳] 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[۵۴] استفاده می‌کند. این مقادیر اطلاعاتی 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.

۱۵. ایده اصلی ابزار AppIntent [۹] 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

[52] Exploit

[53] Selective Record-Replay

[54] Shadow Values

حساس و دنباله رویدادهای مربوط به آنها استخراج می‌شود. 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.

۱۶. در سال ۲۰۱۵ ابزار Sig-Droid [۴] برای آزمون برنامک‌های اندرویدی ارائه شده است. در این ابزار سعی شده است برنامک‌ها روی JVM[۵۵] 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[۵۶] 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.

۱۷. ابزار Condroid [۸] 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 بمب منطقی را در برنامک‌های اندرویدی تشخیص دهد. این ابزار همان مشکلات ACTEVE یعنی انفجار مسیر را به ارث برده است.

۱۸. ابزار Driller [۳۴] 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.
- فازینگ: ابزار ابتدا با فازینگ شروع به کار می‌کند. اگر به ورودی‌های «مشخص» برسد فازر گیر می‌کند.

[55] Java Virtual Machine

[56] Main Method

- اجرای پویا-نمادین : وقتی فازر گیر کرد اجرای پویا-نمادین شروع به کار می‌کند تا مسیر جدیدی را پیدا کند.
- Repeat: وقتی مسیر جدید پیدا شد، اجرا دوباره بهفازر سپرده می‌شود و اجـرا ادامـه پیـدا می‌کند.

یک ویژگی مهم Driller است که وقتی اجرا به موتور پویا-نمادین داده می‌شود، اجرای پویا-نمادین دچار انفجار مسیر نخواهد شد. چون که فازر مسیر اجرای پیشین خـود را بـه موتـور پویـا-نمـادین می‌دهد و اجرای پویا-نمادین تنها سعی می‌کند با مکمل کردن یکی از شرطهای مسیر ورودی ازفازر به مسیری جدیدی برسد.

۲-۵ جمع‌بندی

در این فصل به تفصیل و همراه با مثال اجرای پویا-نمادین مورد بررسی قرار گرفت. در این فصل کارهای صورت گرفته در این حوزه از سال ۲۰۰۵ تا کنون بررسی شدند. همان طور که دیـده مـی‌شـود، اجـرای پویا-نمادین روشی است که می‌تواند به پوشش بالایی از کد برسد. همچنین چون برنامه اجرا مـی‌شـود، مثبت نادرست ندارد. پس می‌تواند روشی مناسب برای تحلیل نرم‌افزارها برای تشخیص آسیب‌پـذیری در آنها باشد. ما از این روش در تحلیل برنامک‌های اندرویدی برای تشخیص آسیب‌پذیری استفاده کرده‌ایـم. همچنین با بهره‌گیری از تحلیل ایستا و ترکیب آن با اجرای پویا-نمادین توانستیم با مشکل انفجار مسیر مقابله کنیم.

۳

فصل سوم

تشخیص آسیب‌پذیری

تشخیص آسیب‌پذیری

یکی از مشکلاتی که در امنیت نرم‌افزارها مطرح است مسئله آسیب‌پذیری‌های موجود در نرم‌افزارهاست. نحوه کشف این آسیب‌پذیری‌ها، از جمله مطالب مورد علاقه محققان بوده است. مطالبی که در قسمت ۳-۳۱-۱ مورد بحث قرار می‌گیرند مطالعه‌ای بر روی آسیب‌پذیری‌های نرم‌افزاری و نحوه کشف آنها به همراه دسته‌بندی روش‌های تشخیص است. همچنین در هر مورد تعدادی ابزار معرفی شده است. پس از آن به بررسی یکی از آسیب‌پذیری‌های معروف یعنی آسیب‌پذیری تزریق خواهیم پرداخت. در ادامه آسیب‌پذیری‌های مطرح در برنامک‌های اندرویدی و روش‌های تشخیص آن‌ها را بررسی می‌کنیم و در آخر به طور خاص آسیب‌پذیری تزریق SQL در برنامک‌های اندرویدی را به تفصیل بررسی خواهیم کرد.

۳-۱ مطالعه‌ای بر روش‌های تشخیص آسیب‌پذیری در نرم‌افزارها

نویسندگان مقاله [35] بیان می‌کنند که تعاریف متفاوتی در مورد آسیب‌پذیری موجود در نرم‌افزارها وجود دارد:

تعریف ۱: ضعف یا خطای موجود در طراحی، پیاده‌سازی و یا اجرای سامانه که یک کاربر بدخواه می‌تواند از آنها در دور زدن خط مشی‌های امنیتی استفاده کند.

تعریف ۲: اشتباه در تعریف[۵۷]، توسعه یا تنظیمات نرم‌افزار که اجرای آن توسط مهاجم به طور صریح یا ضمنی، یکی از خط مشی‌های امنیتی را نقض می‌کند.

تحقیقات در این حوزه در دو دسته طبقه‌بندی می‌شود:

۱. تحلیل آسیب‌پذیری: در این حوزه بر روی ویژگی‌های آسیب‌پذیری‌های موجود تحقیقات انجام می‌شود. مثل دلیل، محل و یا ویژگی‌های پیاده‌سازی. هدف، شناسایی آسیب‌پذیری‌های شناخته نشده است.

[57] Specification

۲. کشف آسیب‌پذیری: هدف، کشف آسیب‌پذیری‌های شناخته شده‌ای است که به صورت ناخواسته در نرم‌افزارها وجود دارند.

در این نوشته تاکید بر روی کشف آسیب‌پذیری است. تکنیک‌های این حوزه به تحلیل ایستا، تحلیل پویا، روش فاز و آزمون نفوذ در نرم‌افزارها تقسیم می‌شود.

۳-۱-۱تحلیل ایستا

تعریف ۳: تحلیل ایستا فرایند ارزیابی یک سیستم بر اساس شکل، ساختار، محتوا یا مستندات آن است و نیازی به اجرای برنامه در آن نیست.

بعضی از آسیب‌پذیری‌ها توسط این روش قابل تشخیص نیست و این یعنی این تحلیل کامل نیست. علاوه بر آن، تحلیل ایستا می‌تواند تخمینی از رفتار برنامه را داشته باشد و این یعنی مثبت نادرست[۵۸] و منفی نادرست[۵۹] در آن بالا است. منفی نادرست خطرناک‌تر از مثبت نادرست است. برای تحلیل مثبت نادرست هم لازم به دخالت انسان است.

یکی از ساده‌ترین ابزارهای موجود در این حوزه، grep در سیستم یونیکس[۶۰] است. با این ابزار رشته‌های موجود در کد برنامه بررسی شده و با لیستی از رشته‌هایی از آسیب‌پذیری‌ها مطابقت داده می‌شود. ابزار دیگر IST4 و RAT است. این دو ابزار از تحلیل کلمات[۶۱] استفاده می‌کنند. در این ابزارها ابتدا یک پیش‌پردازش انجام می‌شود و از کد برنامه کلمات استخراج می‌شوند سپس این کلمات با کلمات موجود در کتابخانه‌ای از کلمات آسیب‌پذیری‌ها مطابقت داده می‌شود. ابزار دیگر SWORD4J است. این ابزار با پیاده‌سازی الگوریتم MARCO (تحلیل SBAC[۶۲]) یا الگوریتم ESPE (تحلیل RBAC[۶۳]) برنامه‌های به زبان جاوا را تحلیل می‌کند.

---

[58] False Posetive

[59] False Negative

[60] Unix

[61] Lexical Analysis

[62] Stack Based Access Control

[63] Role Based Access Control

۳-۱-۲ روش فاز

تعاریف مختلفی از آزمون فاز در مقاله‌های مختلف بیان شده است. از جمله: «روش آزمون با داده‌های دلخواه»، «آزمون جعبه سیاه خودکار» یا «روش آزمون خودکار با داده‌های ورودی مختلف به منظور کشف آسیب‌پذری برنامه». به طور کلی می‌توان این روش را بر اساس مراحل اجرای آن تعریف کرد. ابتدا تولید ورودی دلخواه سپس اجرای برنامه با این ورودی‌ها و در آخر بررسی این که آیا برنامه با این وروی‌ها متوقف می‌شود یا نه.

ابتدا برنامه با داده‌های ورودی کاملا دلخواه اجرا می‌شد که پوشش کاملی از تمام مسیرهای برنامه نداشت. بعد از آن محققان دو روش برای تولید ورودی برنامه‌ها پیشنهاد دادند: ۱) روش تولید-داده ۲) روش جهش-داده. در روش اول، داده‌های ورودی بر اساس تعریف ورودی برنامه ایجاد می‌شود. مثلا اگر ورودی به شکل فایل باشد، ورودی‌ها بر اساس قالب فایل ورودی ایجاد می‌شود. برای این کار نیاز به اطلاعات زیادی در مورد قالب فایل یا پروتکل ورودی است و نیاز به تعامل زیاد انسان است. اگر تعریف داده ورودی پیچیده باشد و تهیه داده نمونه آسان باشد، راه‌کار دوم بهتر است. در این مورد با توجه به ورودی مسیر خاصی از برنامه اجرا می‌شود. ابزارهای Autodafe و APIKE Proxy از روش دوم استفاده می‌کنند. ابزار Peach ترکیب هر دو روش است.

۳-۱-۳ تحلیل پویا

تعریف: خطایابی بر اساس اجرای برنامه.

ویژگی‌های تحلیل پویا

- نیاز به ورودی برای تحلیل برنامه
- تنها خطاهایی که در مسیری که با آن ورودی خاص طی می‌شود قابل شناسایی است
- مثبت نادرست ندارد چون برنامه اجرا می‌شود.

از جمله روش‌های پویا برای تشخیص آسیب‌پذیری اجرای پویا-نمادین است. این روش به تفصیل همراه با ابزارها و نمونه‌ها در فصل پیش شرح داده شد.

۳-۱-۴ آزمون نفوذ

آزمون نفوذ، امنیت یک سیستم را با شبیه‌سازی حمله افراد بدخواه به آن و میزان موفقیت در حمله به آن ارزیابی می‌کند. آزمون نفوذ توسط یک تیم خاص اجرا می‌شود که به استخدام شرکت ارائه دهنده سیستم در می‌آیند و سه دسته کلی دارد:

۱. جعبه سیاه[۶۴]: آزمون نرم‌افزار از بیرون و بدون دسترسی به مستندسازی و مشخصات آن، آزمون جعبه سیاه گویند.
۲. جعبه سفید[۶۵]: در بررسی نرم‌افزارها اگر کد منبع، مستندسازی‌ها و مدل تهدید در اختیار باشد، به آن آزمون جعبه سفید می‌گویند.
۳. جعبه خاکستری[۶۶]: اگر آزمون جعبه سیاه و سفید به صورت ترکیبی استفاده شوند، آزمون جعبه خاکستری خواهیم داشت. در مورد نرم‌افزارهای مختلف، متخصصان یک بار از دید توسعه‌دهندگان و از داخل به بیرون کدها و برنامه را می‌آزمایند. سپس یک بار هم از دید مهاجم و از بیرون به داخل لایه به لایه کدها و برنامه را بررسی می‌کنند.

در آزمون نفوذ دو مرحله وجود دارد. اول آزمون‌گر سیستم را تحلیل کرده و تهدیدات و میزان اثر و اهمیت هر یک را تهیه می‌کند (مثلا درخت تهدید را ایجاد می‌کند). سپس بر اساس اهمیت تهدیدات حملاتی علیه امنیت سیستم اجرا می‌کند. در نهایت در گزارش نهایی علاوه بر اینکه سیستم نسبت به تهدیدی خاص آسیب‌پذیر است یا نه، سناریوهای حمله، نمونه کد حمله و میزان اهمیت آن نیز ذکر خواهد شد.

آسیب‌پذیری‌های مختلفی از جمله سرریزبافر، سرریز هیپ، تزریق کد و غیره مطرح هستند. در این پژوهش ما آسیب‌پذیری تزریق را مورد توجه قرار داده‌ایم. در بخش بعد این آسیب‌پذیری بررسی خواهد شد.

---

[64] Black Box Testing

[65] White Box Testing

[66] Gray Box Testing

۳-۲ آسیب‌پذیری تزریق

در [36] آمده است که وقتی برنامه کاربردی به نوعی باشد که از طرق یک برنامه واسط، یک دستور به سیستم عامل یا پایگاه‌داده ارسال شود، امکان وجود این آسیب‌پذیری وجود خواهد داشت. هرگاه برنامه‌ای از یک مفسر[۶۷] از هر نوعی استفاده کند، خطر اینگونه حملات وجود خواهد داشت.

تعدادی زیادی از برنامه‌های تحت وب از برنامه‌های خارجی یا فراخوانی‌های سیستمی برای ارائه کاربردهای خود استفاده می‌کنند. برنامه Sendmail از پرکاربردترین این برنامه‌هاست. وقتی اطلاعات از طریق یک درخواست HTTP ارسال می‌شود، داده‌های ارسال شده باید کاملا مورد بررسی قرار گیرند. در غیر این صورت مهاجم در میان داده‌ها یک سری دستورات خرابکارانه یا نویسه‌های خاص را ارسال می‌کند. برنامه تحت وب هم کورکورانه این داده‌ها را برای اجرا ارسال می‌کند.

تزریق SQL از جمله انواع شایع حملات تزریق است. در این جا کافی است که مهاجم محلی از برنامه را پیدا کند که داده‌های ورودی را برای اجرای یک پرس‌وجو به پایگاه‌داده ارسال می‌کند. با این کار و جاسازی کردن یک پرس‌وجوی بدخواه در میان داده‌های ارسالی، مهاجم می‌تواند اطلاعات پایگاه‌داده را استخراج کند. اجرای اینگونه از حملات ساده اما بسیار خطرناک هستند و می‌توانند باعث شوند کل سیستم به مخاطره بیفتد.

برای مثال در دستوراتی که به سیستم عامل ارسال می‌شوند می‌توان در ادامه دستورات دیگری نیز اضافه کرد مثلا «; rm –r *» برای پاک کردن تمام فایل‌ها به صورت بازگشتی. یا با اضافه کردن دستوراتی در قسمت «where» در پرس‌وجوهای به پایگاه‌داده می‌توان پرس‌وجوی مورد نظر خود را ارسال و اجرا کرد. مثلا اضافه کردن «” or 1=1» به انتهای پرس‌وجو باعث می‌شود پرس‌وجو به ازای هر مقدار «where» درست باشد.

[67] Interpreter

۳-۲-۱ چه قسمت‌هایی از برنامه می‌توانند آسیب‌پذیر باشند؟

برنامه‌نویس باید کد خود را جست‌وجو کند و تمام محل‌هایی را بررسی کند که به صورت مستقیم (مثلا system, exec, fork, Runtime.exec یا پرس‌وجوهای SQL) یا غیرمستقیم (تمام درخواست‌های HTTP) یک فراخوانی به سیستم یا پایگاه‌داده ارسال می‌شود.

۳-۲-۲ راه‌های مقابله با آسیب‌پذیری تزریق

- استفاده از توابع کتابخانه‌ای موجود به جای فراخوانی‌های سیستمی.
- اعتبارسنجی ورودی توابع خاص و بررسی عملکرد آنها که بدخواه نباشد.
- رعایت اصل حداقل مجوز. یعنی برنامه‌های تحت وب به طور مثال توانایی اجرای یک دستور با مجوز روت[۶۸] را روی سیستم عامل نداشته باشد.
- استفاده از توابع تصفیه‌کننده[۶۹]. مثل بهرمندی از لیست سیاه و سفید برای ورودی کاربر.
- استفاده از توابع پارامتری به جای توابع عادی. در توابع پارامتری اعتبار ورودی کاربر بررسی می‌شود و ورودی به طور مستقیم در محل مشخص شده قرار می‌گیرد. مانند کد شکل ۳-۲.

```
String query = "SELECT account_balance FROM user_data WHERE user_name ="
+  request.getParameter("customerName");
try{
        Statement statement = connection.createStatement;( … );
        ResultSet results = statement.executeQuery( query );
}
```

شکل ۳-۱ نمونه کد ناامن به زبان جاوا برای آسیب‌پذیری تزریق کد SQL

- استفاده از Stored Procedureدها به جای نوشتن پویای پرس‌وجو در رابطه با پایگاه‌داده‌ها.

[68] Root

[69] Sanitizer Methods

```
String custname = request.getParameter("customerName"); // This should REALLY be validated too
// perform input validation to detect attacks
String query = "SELECT account_balance FROM user_data WHERE user_name=?;"
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname) ;
ResultSet results = pstmt.executeQuery;( )
```

شکل ۳-۲ نمونه کد امن به زبان جاوا برای آسیب‌پذیری تزریق کد SQL

در شکل ۳-۱ نمونه کدی به زبان جاوا برای ارتباط با پایگاه داده آمده است. در اینجا رشته پرس‌وجوی ایجاد شده مستقیم با مقدار فراهم آمده از یک درخواست HTTP جمع می‌شود و بدون هیچ بررسی و اعتبارسنجی به تابع executeQuery برای اجرا داده می‌شود. در اینجا مهاجم با تزریق عبارت « '' 1=1#» به عنوان مقدار ارسالی، می‌تواند به تمام مقادیر موجود در جدول user_data دست پیدا کند. این امر زمانی امکان پذیر است که خروجی پرس‌وجو در جایی نمایش داده شود. اما اگر جایی برای نمایش خروجی هم وجود نداشته باشد، باز مهاجم با تزریق «'; DROP TABLE user_data#» می‌تواند با پاک کردن تمام اطلاعات یک جدول، آسیب برساند.

این موضوع اهمیت اعتبارسنجی ورودی کاربر را نشان می‌دهد. در شکل ۳-۲ نمونه امن همان کد آمده است. در اینجا از شکل پارامتری برای گرفتن ورودی از کاربر استفاده می‌شود. در این حالت ورودی کاربر در محل «?» در پرس‌وجو قرار می‌گیرد. معتبر بودن عبارت وارد شده هم توسط کتابخانه مربوط به آن بررسی می‌شود. در این حالت مهاجم نمی‌تواند کد دلخواه خود را تزریق کند.

۳-۳ آسیب‌پذیری در برنامک‌های اندرویدی

در این بخش به طور خاص در مورد آسیب‌پذیری‌های موجود در برنامک‌های اندرویدی و راه‌های تشخیص آنها صحبت می‌کنیم. ابتدا این آسیب‌پذیری‌ها همراه با ابزارهای تشخیص آنها به طور خلاصه عنوان می‌شوند. بعد از آن به طور خاص آسیب‌پذیری تزریق SQL به برنامک‌های اندرویدی و توابع آسیب‌پذیر در آن را همراه با مثال خواهیم دید.

۳-۳-۱ تشخیص آسیب‌پذیری در برنامک‌های اندرویدی

برای تشخیص آسیب‌پذیری در برنامک‌های اندرویدی رویکردهای متفاوتی وجود دارد. تا به حال از روش‌های ایستا بیشتر از روش‌های پویا استفاده شده است. در میان روش‌های پویا تا به حال از روش پویا-نمادین برای تشخیص آسیب‌پذیری استفاده نشده است. در [10] دسته‌بندی کامل و جامعی از تمام روش‌های آزمون امنیتی در حوزه اندروید آمده است. این کار ادعای ما مبنی بر نوع‌آوری ما در استفاده از اجرای پویا-نمادین در تشخیص آسیب‌پذیری را تایید می‌کند. بر اساس نتایج دسته‌بندی این کار تنها ابزارهایی که در فصل پیش به تفصیل توضیح داده شدند از روش پویا-نمادین استفاده می‌کنند و همان طور که گفته شد، تا به حال از این روش در تشخیص آسیب‌پذیری استفاده نشده است. در ادامه تعدادی از کارهایی آورده می‌شود که در حوزه اندروید به منظور تشخیص آسیب‌پذیری ارائه شده‌اند.

ابزار TaintDroid [11] از تحلیل ایستای آلایش به منظور تشخیص آسیب‌پذری نشت اطلاعات حساس کاربر استفاده می‌شود. ابزار SCanDroid [12] هم با تحلیل ایستا به دنبال نشت اطلاعاتی است که خط مشی امنیتی تعریف شده در تنظیمات برنامک را نقض می‌کند.

ابزار CHEX [13] در سال ۲۰۱۲ ارائه شده است این ابزار از روشی ایستا برای تشخیص آسیب‌پذیری «ربودنِ مولفه[۷۰]» استفاده می‌کند. این ابزار با استفاده از گراف جریان داده و گراف وابستگی سیستمی این آسیب‌پذیری را تشخیص می‌دهد.

در مرجع [17]، 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 پژوهش بدون تغییر سیستم عامل به این هدف خواهیم رسید.

در[16] ابزاری ارائه شده است که به وسیله آن می‌توان آسیب‌پذیری برنامک اندرویدی به افشای اطلاعات حساس از طریق Intent را تشخیص داد. در این کار اطلاعات مربوط به Intentها و مجوزهای

[70] Component Hijacking

آنها با تحلیل ایستای فایل Manifest.xml برنامک استخراج می‌شود. سپس در یک محیط شبیه‌ساز برنامک اجرا شده و گردش Intentها دنبال می‌شوند.

در ابزار APSET [14] با روش ایستا به دنبال آسیب‌پذیری‌های مبتنی بر Intent است. این ابزار با استفاده از الگوی آسیب‌پذیری‌ها و دیاگرام کلاس‌های برنامک ویژگی‌های آسیب‌پذیری که باید آزمایش شوند را استخراج می‌کند سپس بر همین اساس برای برنامک مورد نظر مورد آزمون می‌سازد.

در ابزار VulHunter [15] با استفاده از تحلیل ایستا به دنبال تشخیص آسیب‌پذیری‌های نشت توانایی[۷۱]، پیمایش دایرکتوری‌ها در Contet provider[۷۲]، پیاده‌سازی آسیب‌پذیر SSL[۷۳]، مجوز دسترسی عمومی به فایل[۷۴] و ثبت کردن اطلاعات حساس[۷۵] است. در این کار با استفاده از گراف کنترل جریان بین تابعی، گراف وابستگی سیستمی و گراف فراخوانی توابع گراف توصیف برنامه استخراج می‌شود. در این کار آسیب‌پذیری‌های گفته شده مدل شده‌اند و با استفاده از این مدل و گراف توصیف برنامه این آسیب‌پذیری‌ها تشخیص داده می‌شوند.

۲-۳-۳ آسیب‌پذیری تزریق SQL در برنامک‌های اندرویدی

[71] Capability Leak

[72] Content Provider Directory Traversal

[73] X509TrustManager Implemented Improperly

[74] Public File Access Permission

[75] Log Sensitive Information

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 ۳-۳ نمونه برنامه‌ای با شکل استفاده ناامن (خط ۲) و در شکل ۳-۴ استفاده امن (خط ۳) از این تابع‌ها آورده شده است. در شکل ۳-۳ ابتدا ورودی از کاربر توسط editText گرفته می‌شود و به تابع query برای پرس‌وجو داده می‌شود. سپس خروجی در intent ذخیره شده و به secondActivity فرستاده می‌شود. در شکل ۳-۴ نام دانشجو توسط editText گرفته می‌شود و سپس از پایگاه داده پاک می‌شود. در [37] 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 در شکل ۳-۴) با این کار ورودی کاربر به طور مستقیم با دستور برنامه‌نویس جمع نمی‌شود و SDK معتبر بودن ورودی کاربر را بررسی می‌کند. در جدول ۳-۱ مجموعه‌ای از تابع‌های آسیب‌پذیر در SDK آورده شده است. در صورتی

```
1:public void onClick(View v)}
2:  Cursor c = db.query(false, "student", null, "stdno='" + editText.getText().toString()
      + "'", null, null, null, null, nul);
3: Intent myIntent = new Intent(MainActivity.this, SecondActivity.class);
4: int counter = 1;
5: myIntent.putExtra("count", c.getCount());
6: while (c.moveToNext()) {
7:       myIntent.putExtra("name" + counter, c.getString(1));
8:       myIntent.putExtra("stdno" + counter, c.getString(0));
9:       myIntent.putExtra("marks" + counter, c.getString(2));
10:      counter;++
11: }
12: startActivity(myIntent);
```

شکل ۳-۳ نمونه کد آسیب‌پذیر در استفاده از SQLite در اندروید

که درست از آنها استفاده نشود، می‌توانند موجب آسیب‌پذیری تزریق SQL شوند.

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.

جدول ۳-۱ تابع‌های آسیب‌پذیر به تزریق SQL در اندروید

	نام تابع
۱	query
۲	queryWithFactory
۳	rawQuery
۴	rawQueryWithFactory
۵	update
۶	updateWithOnConfilict
۷	delete
۸	execSQL

۳-۴ جمع‌بندی

در این فصل به تفصیل مطالعه‌ای بر روی آسیب‌پذیری‌های نرم‌افزاری انجام دادیم. همچنین آسیب‌پذیری تزریق را به طور خاص و همراه با مثال بررسی کردیم. در ادامه فصل آسیب‌پذیری‌های مربوط به برنامک‌های اندرویدی و روش‌های تشخیص متداول در این حوزه را مطالعه کردیم. با مطالعه روش‌های موجود و کارهای ارائه شده به این نتیجه رسیدیم که تا به حال از روش پویا-نمادین برای تشخیص آسیب‌پذیری در برنامک‌های اندرویدی استفاده نشده است. این روش از آنجایی که می‌تواند به پوشش بالایی از کد برنامه برسد، می‌تواند آسیب‌پذیری‌های مختلف را بهتر تشخیص دهد. چون که

آسیب‌پذیری‌ها از اشتباهات برنامه‌نویس در کدنویسی پدید می‌آیند پس نیاز است تا کد به خوبی تحلیل شود که اجرای پویا-نمادین این مزیت را دارد. همچنین چون برنامک اجرا می‌شود پس مثبت نادرست در تشخیص وجود نخواهد داشت و این موضوع تحلیل را دقیق‌تر می‌کند. البته اجرای پویا-نمادین در برنامک‌های بزرگ در دنیای واقعی با مسئله انفجار مسیر روبه‌رو می‌شود پس نیاز است تا این روش در ترکیب با یک روش ایستا ارائه شود تا نیاز نباشد تا تمام مسیرهای موجود در درخت اجرای برنامک به صورت پویا-نمادین اجرا شوند. در فصل بعد راه‌کاری برای تشخیص آسیب‌پذیری در برنامک‌های اندرویدی با استفاده از اجرای پویا-نمادین پیشنهاد خواهد شد.

```
1:  public void onClick(View view){
2:      Snackbar.make(view, “your record with name=”+ editTextDelete.getText().toString() + “deleted!!”, Snackbar.LENGTH_LONG).setAction(“Action”, null).show;()
3:int a = db.delete(“student”, “name=?”, new String[] { editTextDelete.getText().toString ()});
4:  if (a == 0)
5:      Snackbar.make(view, “your record with name=” + et_del.getText().toString() + “cant be deleted!!”, Snackbar.LENGTH_LONG).setAction(“Action”, null).show;()
6:  else
7:      Snackbar.make(view, “your record with name=” + et_del.getText().toString() + “ deleted!!”, Snackbar.LENGTH_LONG).setAction(“Action”, null).show;()
8:  }
```

شکل ۳-۴ نمونه کد امن در استفاده از SQLite در اندروید

۴

فصل چهارم

راه‌کار پیشنهادی

راه‌کار پیشنهادی

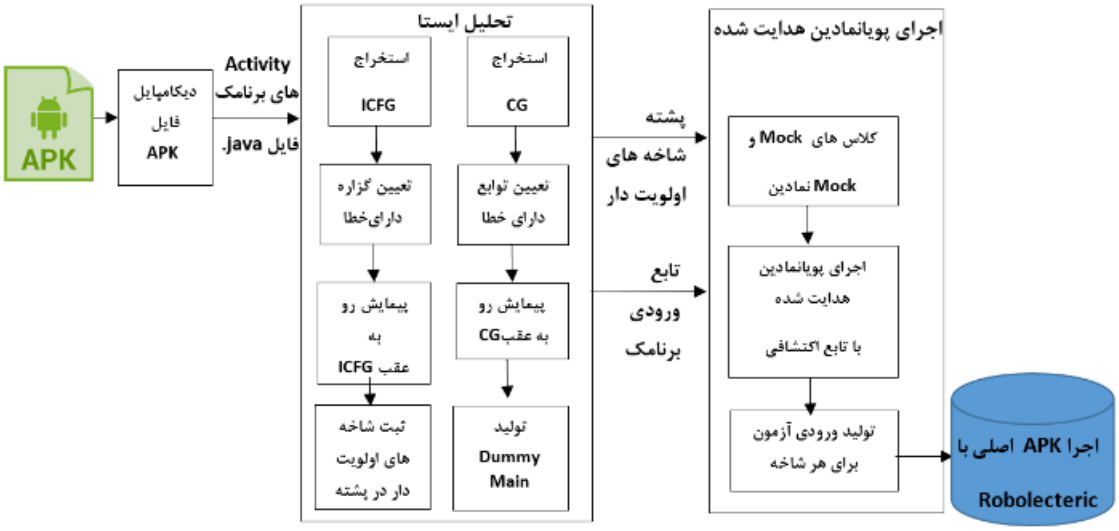
در فصل اول چالش‌های مربوط به تحلیل برنامک‌های اندرویدی توضیح داده شد. در این فصل ما برای مقابله با هر یک و بهینه شدن فرایند تحلیل ایده‌های خود را ارائه خواهیم داد. همچنین در این پژوهش ما بر روی دو موضوع کار کرده‌ایم:

- 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.

۴-۱ ارائه یک هیوریستیک برای اجرای پویا-نمادین هدایت شده

معماری کلی طرح پیشنهادی ما را در شکل ۴-۱ 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 استفاده از خروجی تحلیل ایستا، اجرای پویا-نمادین همراه با هیوریستیک ارائه شده اعمال



شکل ۴-۱ معماری کلی طرح پیشنهادی

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.

۴-۱-۱دیکامپایل 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.

۴-۱-۲ 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[۷۶]» 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[۷۷]»، پشته شاخه‌های اولویت‌دار را تعیین می‌کنیم. این پشته در اجرای هدایت‌شده پویا-نمادین به ما کمک خواهد کرد. هر یک از این دو مورد در ادامه شرح داده خواهند شد.

۴-۱-۲-۱ 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

[76] Call Graph (CG)

[77] Inter- Control Flow Graph (ICFG)

می‌کند. 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.

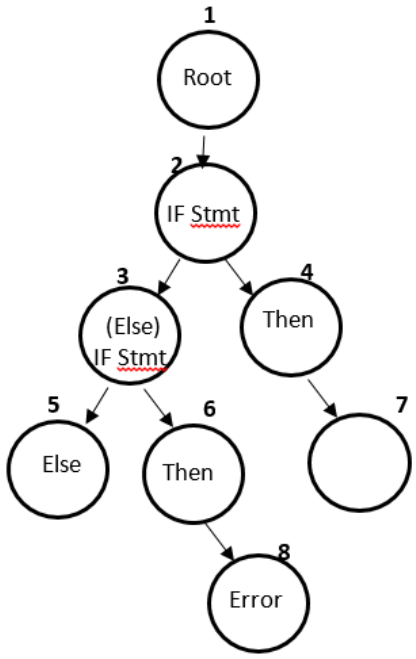
برای تولید تابع نقطه ورودی به برنامه باید گراف فراخوانی توابع را پیمایش نمود. اگر این گراف را به صورت روبه‌جلو و کامل پیمایش کنیم، می‌توانیم به حداکثر پوشش کد دست یابیم. اما با توجه به اینکه یافتن خطا مهمتر از پوشش حداکثری کد است، ما در اینجا ایده پیمایش روبه‌عقب گراف فراخوانی توابع، از تابع دارای خطا به ریشه را مطرح می‌کنیم. گراف فراخوانی توابع را با ابزار Soot [38] به دست آورده‌ایم. سپس الگورتیم پیمایش رو به عقب پیشنهادی خودمان را روی گراف استخراج شده اعمال

```
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 ۴-۲ نمونه تابع نقطه شروع برنامک برای برنامه MunchLife

کرده‌ایم. نمونه تابع نقطه شروع به برنامک در شکل ۴-۲ دیده می‌شود.

۴-۱-۲-۲ تعیین پشته شاخه‌های اولویت‌دار



شکل ۴-۳ مثالی از گراف کنترل جریان بین

برای به دست آوردن 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 رو به عقب 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. سپس این پشته را به عنوان ورودی به اجرای پویا-نمادین خواهیم داد.

برای مثال در شکل ۴-۳ 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 شاخهthen بر else اولویت دارد را در پشته 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.

۴-۱-۳ 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 فصل اول 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 کـه از کـاربر 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. برای مثـال کلاس Mock مرتبط با EditText در شکل ۴-۴ آمده است. همان طور کـه در شـکل دیـده مـی‌شـود

```
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 ۴-۴ نمونه کلاس Mock نمادین تولید شده برای دریافت ورودی نمادین.

خطوطی که پررنگ نشان داده شده‌اند، موجب شده‌اند که کلاس EditText بـه صـورت Mock نمـاین تولید شود. رشته Content در تابع سازنده[۷۸] به صورت رشته نمادین مقداردهی اولیـه مـی‌شـود. همـین مقدار نمادین در تابع getText باز می‌گردد. همان طور که دیده می‌شـود بـا ایـده Mock نمـادین لازم

[78] Constructor

نیست تا تعامل کاربر برای وارد کردن ورودی نمادین وجود داشته باشد و این موضوع چالش رخدادمحور بودن در برنامک‌های اندرویدی را حل می‌کند.

۴-۱-۴ 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 قسمت از 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 تابع‌های نقطه شروعی که به خطا منتهی می‌شوند، محدود می‌کنیم.

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

ایده الف را در بخش ۴-۱-۲-۱استخراج نقطه ورودی برنامه۴-۱-۲-۱ و ایده ب را در بخش تعیین پشته شاخه‌های اولویت‌دار شرح داده‌ایم. برای اجرای پویا-نمادین از SPF استفاده کرده‌ایم. در SPF به صورت پیش فرض 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 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.

۴-۱-۵ 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 [20] 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 ۴-۵ آمده است. همان طور که دیده می‌شود خط ۳ از شکل ۴-۲ مشابه خط ۲ از شکل ۴-۵ است. خطوط۴ و ۵ از شکل ۴-۲ به صورت خودکار در ابزار اجرا می‌شوند.

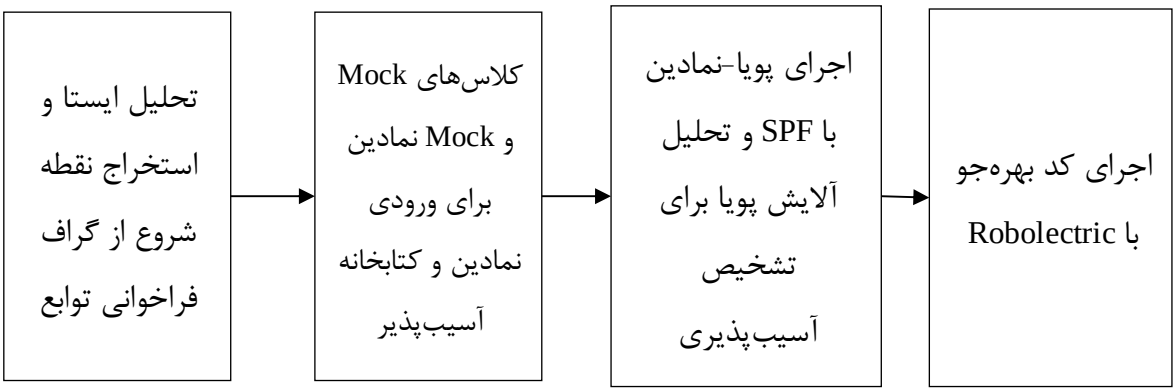
```
1: public void TestofApp() throws Exception {
2:        Activity ma = Robolectric.setupActivity(MunchLifeActivity.class);
3:}
```

شکل ۴-۵ نمونه کد آزمون در Robolectric برای برنامه MunchLife.

۴-۲ اجرای پویا-نمادین برای تشخیص آسیب‌پذیری تزریق

در شکل ۴-۶ 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 ۴-۶ 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 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



شکل ۴-۶ فرایند تشخیص آسیب‌پذیری در ابزار

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. 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.

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 داده‌ایم.

۴-۲-۱ 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 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 ۴-۷)

```
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 ۴-۷ نمونه dummyMain تولید شده برای اجرا در SPF

۴-۲-۲ 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 برنامک‌های دیگر حذف شود. چون کلاس‌های منبع و آسیب‌پذیر هر دو بخشی از SDK هستند، پس از این جهت به شکل Mock پیاده‌سازی می‌شوند. 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. نمونه کلاس Mock نماین برای تابع منبع در شکل ۴-۴ آمده است. بخشی از کلاس Mock نماین برای تابع آسیب‌پذیر کلاس SQLiteDatabase

```
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 ۴-۸ تکه کدی از کلاس Mock نمادین تولید شده برای کلاس SQLiteDatabase

در شکل ۴-۸ دیده می‌شود. همان طور که دیده می‌شود تمام خروجی‌های تابع‌ها نمادین تولید می‌شوند. کد کامل این کلاس در پیوست آمده است.

۴-۲-۳ 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 آسیب‌پذری 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) برسیم. در صورتی که ورودی تابع نشت، از

```
----------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 ۴-۹ نمونه خروجی ابزار برای تشخیص آسیب‌پذیری تزریق SQL

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. در شکل ۴-۹ نمونه خروجی تولید شده برای یک برنامک آسیب‌پذیر آمده است.

۴-۲-۴ 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 ۴-۱۰ نمونه کد بهرجو در Robolectric 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 ۴-۷ با ۲ شکل ۴-۱۰ نمونه کد بهرجو در Robolectric، خط ۵ شکل ۴-۷ با ۳ شکل ۴-۱۰ نمونه کد بهرجو در Robolectric و ۶ شکل ۴-۷ با ۷ شکل ۴-۱۰ نمونه کد بهرجو در Robolectricیکی 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 ۴-۱۰ نمونه کد بهرجو در Robolectric) برای بررسی بهره‌جویی، خروجی تابع نشت یعنی TextView را مشاهده کردیم (خط ۸ شکل ۴-۱۰ نمونه کد

```
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 ۴-۱۰ نمونه کد بهرجو در Robolectric

بهرجو در Robolectric). برای یافتن شناسه تابع ورودی آسیب‌پذیر و شناسه تابع نشت از خروجی تحلیل پویا-نمادین استفاده کردیم.

۴-۳ جمع‌بندی

در این فصل راه‌کار پیشنهادیمان را در رابطه با تشخیص آسیب‌پذیری در برنامک‌های اندرویدی با اجرای پویا-نمادین شرح دادیم. برای این کار دو سوال پژوهشی مطرح کردیم. در سوال اول به دنبال بهبود اجرای پویا-نماین در آزمون برنامک‌های اندرویدی بودیم. برای حل این موضوع از تحلیل ایستای برنامک و گراف فراخوانی توابع و گراف کنترل جریان بین تابعی و ایده پشته شاخه‌های اولویت دار به منظور هدفمندکردن اجرای پویا-نمادین استفاده کردیم. در سوال پژوهشی دوم به دنبال اجرای پویا-نمادین برای تشخیص آسیب‌پذری تزریق در برنامک‌های اندرویدی بودیم. برای این موضوع از ترکیب تحلیل آلایش با اجرای پویا-نمادین استفاده کردیم. همچنین برای بهبود هزینه و سربار، تحلیل را محدود به توابع مطلوبی از برنامک کردیم که از تحلیل ایستا و گراف فراخوانی توابع بدست آورده بودیم. همچنین ایده Mock نمادین را مطرح کردیم که باعث گردش داده‌های آلایش‌شده در برنامه می‌شود. با ترکیب این موارد زنجیره تشخیص آسیب‌پذیری تزریق یعنی دنبال کردن داده‌ها از تابع منبع به تابع آسیب‌پذیر و از تابع آسیب‌پذیر تا تابع نشت را کامل کردیم. به عنوان خروجی شناسه تابع منبع، شناسه تابع نشت، دنباله پشته برنامه تا تابع آسیب‌پذیر و تصفیه شدن یا نشدن داده ورودی توسط برنامه‌نویس را ارائه کردیم. در فصل بعد راه‌کار پیشنهادی را مورد آزمون قرار می‌دهیم و نتایج آزمایش‌ها و تحلیل آنها را ارائه می‌کنیم. در نهایت روش خود را با آخرین کارهای مرتبط موجود در این حوزه مقایسه خواهیم کرد.

۵

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[۷۹]» 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 آلایش‌شده در تحلیل آلایش) و اجرای پویا-نمادین برنامک به این هدف رسیدیم. در این کار از ایده Mock نمادین برای توابع آسیب‌پذیر استفاده 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.

۵-۱ ارزیابی هیوریستیک ارائه‌شده 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

[79] ICFG

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 ۵-۱ را آزمودیم که برنامک‌های مورد آزمون ابزار Sig-Droid نیز هستند. این برنامک‌ها از مخزن F-Droid [21] انتخاب شده‌اند. در جدول ۵-۱ 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 می‌شوند. 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 می‌شود. نویسنده عنوان می‌کند با هدف پوشش کد هر چه بیشتر این ابزار پیاده‌سازی شده است. در نهایت این ابزار با ابزارهای مطرحی چون Monkey و Dynodroid [39] 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 ۵-۱ 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 ۵-۲ اطلاعات مربوط به تحلیل برنامک‌های جدول ۵-۱ 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 است که در این تحلیل اگر ما اجرای پویا-نمادین را بدون اولویت‌گذاری شاخه‌ها اجرا کنیم و همچنین همه تابع‌های تقطه شروع را مورد آزمون قرار دهیم و علاوه‌بر آن به جای اجرای پویا-نمادین برنامه را به صورت نمادین اجرا کنیم، کار ما همان Sig-Droid خواهد بود و نتایج مشابه مقاله آن ابزار خواهد شد.

جدول ۵-۲ 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 ۵-۳، مقایسه ابزارهای مختلف با کار ما بر اساس معیارهـای موجـود در مقـالات [3] [4] [7] آمده است. در معیارهای مقایسه انتخاب شده به ترتیب روش جست‌وجو، انواع رخدادهای پشتیبانی شده در ابزار، ترکیب تحلیل ایستا و پویا به عنوان هیوریستیک در بهینه سازی اجـرا و چـالش انفجـار مسـیر مطرح شده‌اند. همان طور که پیش از این گفته شد برای آزمون برنامک‌های اندرویـدی ۳ روش متـداول 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[۸۰] (مانند فشردن یک دکمه) قرار می‌گیرند. لازم بـه ذکـر اسـت کـه مسـئله انفجـار مسـیر در Monkey و Swifthand مطرح نمی‌شود چون این دو ابزار مسیرهای مختلف موجود در کد را بررسی و اجرا نمی‌کنند.

همان طور که در جدول هم دیده می‌شود، 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

[80] GUI

نمادین می‌تواند به پوشش بهتری از کد برسد. در کار ما بـا اسـتفاده از ایـده Mock و Mock نمـادین می‌توانیم انواع رخدادهای مطرح را پشتیبانی کنیم. با استفاده از هیوریسـتیک ارائـه شـده کـه ترکیـب تحلیل ایستا و پویا هست توانستیم با سرعت بیشتر و پوشش کـد کمتـر خطاهـا را پیـدا کنـیم، همـین موضوع موجب می‌شود که با چالش انفجار مسیر روبه‌رو نشویم.

جدول ۵-۳ 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	مبتنی بر مدل	متن،GUI	×	-
Sig-Droid	نمادین	متن،GUI	×	×
کار ما	پویا-نمادین	متن،سیستم،GUI	✓	✓

۵-۲ ارزیابی تشخیص آسیب‌پذیری تزریق در برنامک‌های اندرویدی

ارزیابی راه‌کار ارائه شده در این بخش را در دو مرحله انجام دادیم. ابتدا برای نشان دادن ایـن کـه روش درست کار می‌کند ۱۰ برنامک را خودمان پیاده‌سازی کردیم. ۶ برنامک را به منظور راستی‌آزمایی تولیـد کلاس dummyMain و تابع نقطه شروع، همچنین درست بـودن فراینـد تولیـد کـلاس‌هـای Mock و درست بودن اجرای پویا-نمادین و ارتباط این مولفه‌ها با هم پیاده‌سازی کردیم. در این برنامک‌هـا بـرای 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 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 نیز 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 تابع نقطه شروع بسازیم. از این برنامک‌ها، برای ۷ برنامک، تابع نقطه شروع 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. 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 ۵-۴ مقایسه ابزار ما با ابزارهای مشابه فعلی از ۵ جنبه مطرح در مقالات آمده است[4][8][9] . ابزارهای Condroid و AppIntent که در این جدول با کار ما مقایسه شده‌اند دغدغه امنیتی دارند. Condroid با استفاده از اجرای پویا-نمادین به دنبال کشف بمب منطقی در برنامک‌های اندرویدی است. AppIntent با اجرای نمادین به دنبال کشف نقض حریم خصوصی توسط 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 شد، به منظور آزمون برنامک‌ها و رسیدن به پوشش 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 می‌شوند 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 ۵-۴ 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	✓	✓	✓	✗	✓	✗

۵-۳ جمع‌بندی و کارهای آینده

در این پژوهش ما به دنبال اجرای پویا-نمادین برای تشخیص آسیب‌پذیری تزریق به برنامک‌های گوشی‌های هوشمند بودیم. از میان سیستم عامل‌های مختلف مربوط به گوشی‌های هوشمند، اندروید را برگزیدیم چون که هم از محبوبیت بالایی برخوردار است و هم متن‌باز بوده و می‌توان به کتابخانه‌ها و کدهای آن دسترسی داشت. ابتدا چالش‌های مربوط به آزمون برنامک‌های اندرویدی و تفاوت‌های آن را بررسی کردیم. برای تحلیل و آزمون، اجرای پویا-نمادین را انتخاب کردیم چون علاوه بر پوشش قوی کد که برای تشخیص آسیب‌پذیری نیاز است، مثبت نادرست هم ندارد. همان طور که عنوان شد، موتور اجرای پویا-نمادین برای برنامک‌های اندرویدی موجود نیست. ما در این کار از SPF استفاده کردیم که موتور اجرای پویا-نمادین برنامه‌های جاوا است.

همان طور که گفته شد، استفاده از SPF برای آزمون برنامک‌های اندرویدی چالش‌های مرتبط به خود را دارد. از جمله وجود SDK و درهم‌تنیدیگی زیاد برنامک با آن، وجود نداشتن نقطه شروع به برنامه، کامپایل شدن روی JVM برخلاف DVM و هم چنین رخدادمحور بودن برنامک‌های اندرویدی. برای هر یک از این چالش‌ها راه‌کار مرتبط با آن را از جمله راه‌کار Mock و Mock نمادین و تحلیل ایستا و پیمایش گراف فراخوانی توابع برای یافتن نقطه شروع به برنامه، ارائه دادیم.

در روند پژوهشی خود به دو سوال پژوهشی پاسخ دادیم. ابتدا راهکاری برای بهبود اجرای پویا-نمادین برای کشف خطا و تولید خودکار ورودی آزمون ارائه کردیم و در ادامه راه‌کاری برای تشخیص آسیب‌پذیری تزریق به برنامک‌های اندرویدی را پیشنهاد دادیم. هر کدام از این سوالات را مورد ارزیابی قرار دادیم.

نتیجه ارزیابی برای راه‌کار ارائه شده سوال اول این است که با سرعت بیشتر و پوشش کمتر کد نسبت به Sig-Droid که آخرین کار در این مورد است، خطا را تشخیص می‌دهیم. 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.

در مورد سوال دوم هم در مجموع ۱۵۰ برنامک را مورد تحلیل قرار دادیم که ۱۱ مورد آن آسیب‌پذیر به تزریق SQL بود و توانستیم همه را تشخیص دهیم. 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.

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 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 کنیم. لازم به ذکر است که گراف فراخوانی توابع استخراج شده از ابزار Soot، وابستگی کلاس‌ها به واسطه Intent را در خود ندارد. در آینده این ابزار را برای این منظور بهبود خواهیم داد.

۶

منابع و مراجع

[1] "Cafebazaar." [Online]. Available: http://developers.cafebazaar.ir/fa/. [Accessed: 10-Nov-2017].

[2] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, p. 1, 2012.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "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.

[10] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A Taxonomy and

Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–48, 2016.

[11] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, p. 5, 2014.

[12] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android," 2009.

[13] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 229--240.

[14] S. Salva and S. R. Zafimiharisoa, "APSET, an Android aPplication SEcurity Testing tool for detecting intent-based vulnerabilities," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 2, pp. 201--221, 2015.

[15] C. Qian, X. Luo, Y. Le, and G. Gu, "Vulhunter: toward discovering vulnerabilities in android applications," *IEEE Micro*, vol. 35, no. 1, pp. 44--53, 2015.

[16] A. Avancini and M. Ceccato, "Security testing of the communication among Android applications," in *Automation of Software Test (AST), 2013 8th International Workshop on*, 2013, pp. 57--63.

[17] G. J. Smith, "Analysis and Prevention of Code-Injection Attacks on Android OS," University of South Florida, 2014.

[18] "OWASP Mobile Security Testing Guide - OWASP." [Online]. Available: https://www.owasp.org/index.php/OWASP_Mobile_Security_Testing_Guide. [Accessed: 09-Nov-2017].

[19] C. S. Pasareanu and N. Rungta, "Symbolic PathFinder: Symbolic Execution of Java Bytecode," *25th IEEE/ACM International Conference on Automated Software Engineering*, vol. 2, pp. 179–180, 2010.

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

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

[22] 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.

[23] 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.

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

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

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

[27] 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.

[28] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jFuzz: A concolic whitebox fuzzer for Java," 2009.

[29] 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.

[30] 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.

[31] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[32] 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.

[33] 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.

[34] 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.

[35] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," *Proceedings - 2012 4th International Conference on Multimedia and Security, MINES 2012*, pp. 152–156, 2012.

[36] "OWASP-Injection Flaw." [Online]. Available: https://www.owasp.org/index.php/Injection_Flaws. [Accessed: 01-Nov-2018].

[37] "SQL Injection - OWASP." [Online]. Available: https://www.owasp.org/index.php/SQL_Injection. [Accessed: 09-Nov-2017].

[38] 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.

[39] 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.

.

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 of Computer Engineering and Information Technology

MSc Thesis

Concolic Execution for Detecting Injection Vulnerabilities in Mobile Apps

By
Ehsan Edalat

Supervisor
Dr. Babak Sadeghiyan

December 2017