



دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)
دانشکده مهندسی کامپیوتر و فناوری اطلاعات

پایان نامه کارشناسی ارشد
گرایش امنیت اطلاعات

اجرای پویا-نمادین برای تشخیص آسیب پذیری تزریق به برنامه های
کاربردی گوشه های هوشمند

نگارش
احسان عدالت

استاد راهنما
جناب آقای دکتر بابک صادقیان



دانشگاه صنعتی امیر کبیر

دانشکده مهندسی کامپیوتر و فناوری اطلاعات

پایان نامه کارشناسی ارشد

گرایش امنیت اطلاعات

اجرای پویا-نمادین برای تشخیص آسیب پذیری تزریق به برنامه‌های کاربردی گوشی‌های هوشمند

نام و نام خانوادگی: احسان عدالت شماره دانشجویی: ۹۴۱۳۱۰۹۰ مقطع: کارشناسی ارشد

این پایان نامه توسط هیئت داوران زیر در تاریخ ۱۳۹۶ / ۱۱ / ۲۵ به تصویب رسیده است:

امضا:

استاد راهنما: دکتر بابک صادقیان

امضا:

داور داخلی: مهرا ن سلیمان فلاح

امضا:

داور خارجی: دکتر مهدی آبادی



دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

به نام خدا

تعهدنامه اصالت اثر

تاریخ: ۱۳۹۷/۵/۶

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

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 سلیمان فلاح و آقای دکتر آبادی 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، احمد اسدی و حمیدرضا رضانی که در طول پروژه از راهنمایی‌هایشان استفاده کردم؛

کمال تشکر و قدردانی را دارم.

چکیده

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. 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 برنامه‌ها (برنامه‌کاربردی)های اندرویدی نسبت به برنامه‌های دیگر دارای چالش‌های جدید رخداده‌محور بودن و وابستگی زیاد به SDK است که سربرار آزمون را بالا می‌برد. همچنین این برنامه‌ها برخلاف برنامه‌های به زبان جاوا، نقطه شروع مشخصی ندارند. لازم به ذکر است که تا به حال موتور اجرای پویا-نمادین برای این برنامه‌ها ارائه نشده است. در این پژوهش ما با استفاده از موتور SPF، روشی برای تشخیص آسیب‌پذیری تزریق در برنامه‌های اندرویدی ارائه کرده‌ایم.

در این پژوهش ما یک روش هیوریستیک را ارائه کرده‌ایم که اجرای پویا-نمادین را به صورت بهینه و هدایت شده روی برنامه‌های 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 سربرار بالای آزمون برنامه‌ها را کاهش دهیم. برای ارزیابی، علاوه بر ۱۰ برنامه پیاده‌سازی‌شده خودمان، ۴ برنامه 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 برنامه را تشخیص دهد.

همچنین، در این پژوهش روشی ارائه می‌شود که با اجرای پویا-نمادین همراه تحلیل آرایش به دنبال تشخیص آسیب‌پذیری تزریق در برنامه‌های اندرویدی هستیم. در این کار با تحلیل ایستا، گراف فراخوانی توابع و پیمایش برعکس از تابع آسیب‌پذیر تا تابع منبع، نقطه شروع برنامه 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 رخداده‌محور بودن و سربرار بالای آزمون برنامه‌ها را حل کرده‌ایم. برای ارزیابی، علاوه بر ۱۰ برنامه پیاده‌سازی‌شده خودمان، 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.

واژه‌های کلیدی:

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

فهرست مطالب

۱ فصل اول مقدمه.....	۱
۲ فصل دوم اجرای پویا-نمادین.....	۱۲
۱-۲ بیان اجرای نمادین و پویا-نمادین با مثال.....	۱۳
۲-۲ چالش‌های اجرای پویا-نمادین.....	۱۵
۲-۳ انواع اجرای پویا-نمادین.....	۱۶
۴-۲ کارهای گذشته.....	۱۷
۵-۲ جمع‌بندی.....	۲۷
۳ فصل سوم تشخیص آسیب‌پذیری.....	۲۸
۱-۳ مطالعه‌ای بر روش‌های تشخیص آسیب‌پذیری در نرم‌افزارها.....	۲۹
۱-۱-۳ تحلیل ایستا.....	۳۱
۲-۱-۳ روش فازینگ.....	۳۲
۳-۱-۳ تحلیل پویا.....	۳۲
۴-۱-۳ آزمون نفوذ.....	۳۳
۲-۳ آسیب‌پذیری تزریق.....	۳۴
۱-۲-۳ قسمت‌های آسیب‌پذیر برنامه.....	۳۶
۲-۲-۳ راه‌های مقابله با آسیب‌پذیری تزریق.....	۳۶
۳-۳ آسیب‌پذیری در برنامه‌های اندرویدی.....	۳۸
۱-۳-۳ آسیب‌پذیری‌های تزریق در برنامه‌های اندرویدی.....	۳۸
۲-۳-۳ تشخیص آسیب‌پذیری در برنامه‌های اندرویدی.....	۴۰
۳-۳-۳ آسیب‌پذیری تزریق SQL در برنامه‌های اندرویدی.....	۴۲
۴-۳ جمع‌بندی.....	۴۴
۴ فصل چهارم راه‌کار پیشنهادی.....	۴۶
۱-۴ ارائه یک روش هیوریستیک برای اجرای پویا-نمادین هدایت شده.....	۴۷
۱-۱-۴ دیکامپایل برنامه.....	۴۸
۲-۱-۴ تحلیل ایستا.....	۴۹
۱-۲-۴ استخراج نقطه ورودی برنامه.....	۵۰
۲-۲-۴ تعیین پشته شاخه‌های اولویت‌دار.....	۵۱
۳-۱-۴ تولید کلاس‌های Mock و Mock نمادین.....	۵۲
۴-۱-۴ اجرای پویا-نمادین هدایت شده با روش هیوریستیک.....	۵۴
۵-۱-۴ اجرای برنامه با ورودی‌های عینی.....	۵۶
۲-۴ اجرای پویا-نمادین برای تشخیص آسیب‌پذیری تزریق.....	۵۶

۵۸.....	تحلیل ایستا.....	۱-۲-۴
۵۹.....	تولید کلاس‌های Mock و Mock نمادین.....	۲-۲-۴
۶۰.....	اجرای پویا-نمادین همراه با تحلیل آرایش.....	۴-۲-۳
۶۲.....	آزمون نرم‌افزار برای بررسی میزان بهره‌جویی.....	۴-۲-۴
۶۳.....	جمع‌بندی.....	۳-۴
۶۴.....	فصل پنجم ارزیابی و جمع‌بندی.....	۵
۶۵.....	ارزیابی روش هیوریستیک ارائه‌شده برای اجرای هدایت شده پویا-نمادین.....	۱-۵
۷۰.....	ارزیابی تشخیص آسیب‌پذیری تزریق در برنامه‌های اندرویدی.....	۲-۵
۷۳.....	جمع‌بندی و کارهای آینده.....	۳-۵
۷۵.....	منابع.....	۷۵
۷۹.....	پیوست الف.....	۷۹
۸۲.....	پیوست ب.....	۸۲

صفحه

فهرست اشکال

شکل ۱-۲ نمونه برنامه ساده.....	۱۳
شکل ۲-۲ درخت اجرای اجرای نمادین برنامه نمونه.....	۱۴
شکل ۱-۳ شمای کلی از برنامه‌های دارای پایگاه داده و ارتباط مولفه‌های مختلف با هم.....	۳۵
شکل ۲-۳ نمونه کد ناامن به زبان جاوا برای آسیب‌پذیری تزریق کد SQL.....	۳۷
شکل ۳-۳ نمونه کد امن به زبان جاوا برای آسیب‌پذیری تزریق کد SQL.....	۳۷
شکل ۴-۳ نمونه کد آسیب‌پذیر به تزریق دستور پوسته سیستم عامل.....	۳۹
شکل ۵-۳ نمونه کد برای آسیب‌پذیری تزریق جاوا اسکریپت به WebView.....	۳۹
شکل ۶-۳ نمونه کد آسیب‌پذیر در استفاده از SQLite در اندروید.....	۴۳
شکل ۷-۳ نمونه کد امن در استفاده از SQLite در اندروید.....	۴۳
شکل ۱-۴ معماری کلی طرح پیشنهادی.....	۴۸
شکل ۲-۴ خروجی دیکامپایل برای یک برنامه نمونه.....	۴۹
شکل ۳-۴ نمونه تابع نقطه شروع شروع برنامه برای MunchLife.....	۵۰
شکل ۴-۴ مثالی از گراف کنترل جریان بین تابعی.....	۵۲
شکل ۵-۴ نمونه کلاس Mock نمادین تولید شده برای دریافت ورودی نمادین.....	۵۳
شکل ۶-۴ نمونه کد آزمون در Robolectric برای برنامه MunchLife.....	۵۶
شکل ۷-۴ فرایند کاری مطابق با معماری کلی طرح پیشنهادی برای تشخیص آسیب‌پذیری تزریق در برنامه‌های اندرویدی.....	۵۷
شکل ۸-۴ نمونه dummyMain تولید شده برای اجرا در SPF.....	۵۹
شکل ۹-۴ تکه کدی از کلاس Mock نمادین تولید شده برای کلاس SQLiteDatabase.....	۶۰
شکل ۱۰-۴ نمونه خروجی ابزار برای تشخیص آسیب‌پذیری تزریق SQL.....	۶۱
شکل ۱۱-۴ نمونه کد بهره‌بردار در Robolectric.....	۶۲

صفحه

فهرست جداول

جدول ۱-۲ کارهای گذشته.....	۱۷
جدول ۱-۳ تابع‌های آسیب‌پذیر به تزریق SQL در اندروید.....	۴۴
جدول ۱-۵ مشخصات برنامه‌های دنیای واقعی مورد آزمون.....	۶۷
جدول ۲-۵ نتایج بدست آمده با ابزار ما و Sig-Droid پیاده‌سازی شده توسط ما.....	۶۸
جدول ۳-۵ مقایسه ابزارهای مختلف با کار ما.....	۶۹
جدول ۴-۵ مقایسه ابزار ارائه شده با ابزارهای مشابه موجود.....	۷۲

۱

فصل اول

مقدمه

مقدمه

میزان آگاهی برنامه نویسان از نحوه توسعه امن نرم افزار یکسان نیست. این مورد باعث می شود نرم افزارهای تولید شده با تهدیدهای امنیتی نیز در بازار قرار گیرد که موجب نشت اطلاعات حساس کاربران و یا نقض حریم خصوصی آنها می شود. از این رو نیاز به وجود راه کارهایی برای خودکار کردن فرایند تشخیص وجود آسیب پذیری در نرم افزارها احساس می شود. در سال های اخیر ارائه و توسعه ابزارهای موبایل گسترش پیدا کرده است. از میان سیستم عامل های استفاده شده در این ابزارها می توان به iOS، اندروید و ویندوز اشاره کرد. حجم زیادی از این ابزارها مبتنی بر سیستم عامل اندروید هستند. اندروید محبوب ترین سیستم عامل حال حاضر ابزارهای موبایل است. با گسترش اندروید، توسعه برنامه های اندرویدی نیز رشد چشمگیری داشته اند به طوری که فقط در فروشگاه داخلی کافه بازار، تاکنون بیش از ۱۴۰ هزار برنامه اندرویدی برای بیش از ۳۵ میلیون مخاطب داخلی منتشر شده است. [۱] این موضوع باعث شده است که پژوهش های زیادی در رابطه با برنامه های اندرویدی صورت بگیرد. همچنین سیستم عامل اندروید برخلاف iOS و ویندوز متن باز است. این مورد کار پژوهش بر روی این سیستم عامل را راحت تر می کند.

گوگل برای آسان کردن فرایند توسعه نرم افزار مجموعه ای از ابزار و کد یعنی SDK^۱ را ارائه داده است. برنامه های اندرویدی با اضافه کردن کدهای برنامه نویسی به SDK تولید می شوند. این برنامه ها از جمله برنامه های رخدادمحور محسوب می شوند. تفاوت عمده آنها با سایر برنامه ها، در هم تنیدگی زیاد با SDK است. این موضوع باعث می شود برای اجرای یک قطعه کد ساده برنامه نویسی، تعداد زیادی از قطعه کدهای SDK فراخوانی و اجرا شوند و این موضوع خودکار کردن فرایند آزمون و تشخیص آسیب پذیری را با چالش روبه رو می کند. به طور معمول برنامه های اندرویدی با زبان جاوا پیاده سازی می شوند. این موضوع معمولاً باعث به وجود آمدن این تصور می شود که برنامه های اندرویدی با برنامه های به زبان جاوا که برای پلتفرم هایی مثل کامپیوتر شخصی پیاده سازی می شوند، تفاوتی ندارند. اما این تصور درستی

^۱ Software Development Kit

نیست. در زیر تفاوت‌های عمده اجرای نمادین و پویا-نمادین^۲ برنامه‌های 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 برنامه‌های اندرویدی پیاده‌سازی نشده است. دلیل این موضوع هم چالش‌های موجود در آزمون این برنامه‌ها است. در این پژوهش ما از موتورهای اجرای پویا-نمادین برنامه‌های به 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 برنامه‌های اندرویدی با سایر برنامه‌ها را می‌توان در سه مورد عنوان کرد:

۱. برنامه‌های اندروید وابسته به مجموعه‌ای از کتابخانه‌ها هستند که در بیرون از دستگاه یا شبیه‌ساز^۳ در دسترس نیستند. کد اندروید در DVM^۴ اجرا می‌شود. برخلاف کدهای برنامه‌های جاوا که در JVM^۵ اجرا می‌شوند. پس به جای بایت‌کد جاوا برنامه‌های اندرویدی به بایت‌کد Dalvik کامپایل می‌شوند. برنامه‌های به زبان جاوا همگی دارای «تابع نقطه شروع»^۶ به برنامه هستند. یعنی این برنامه‌ها بدون استثنا از این تابع شروع به اجرا می‌شوند. ولی برنامه‌های اندرویدی چنین تابعی ندارند و اجرای یک برنامه به شکل‌های مختلفی امکان اجرا دارد. برای مثال ممکن است یک برنامه با بازکردن مستقیم آن شروع شود. در حالی دیگر ممکن است این برنامه با اتفاق افتادن یک رخداد به خصوص، مثلا دریافت یک پیامک، شروع به اجرا کند. این موضوع خود چالشی جدی در تحلیل و آزمون این برنامه‌ها با موتور اجرای پویا-نمادین برای برنامه‌های به زبان جاوا خواهد بود. همچنین این برنامه‌ها باید به نحوی تغییر پیدا کنند که بتوان آنها را در JVM کامپایل کرد.

^۲ پویا-نمادین: Concolic عبارتی مخفف و معادل با Concrete + Symbolic می‌باشد. کلمه Concrete در ترجمه فارسی معادل با کلمات ماده، جوهر و یا ذات ترجمه می‌شود. از آن جا که در نوشته‌جات عبارت Dynamic Symbolic به عنوان معادل برای روش Concolic استفاده نیز می‌گردد، ما برای لغت معادل فارسی از عبارت پویا-نمادین استفاده کرده‌ایم.

^۳ Emulator

^۴ Dalvik Virtual Machine

^۵ Java Virtual Machine

^۶ Main Method

۲. برنامه‌های اندرویدی بسیار وابسته به کتابخانه‌های چارچوبه کاری یعنی SDK هستند و این موضوع باعث ایجاد مشکل واگرایی مسیر^۷ [۲] می‌شود. در اجرای نمادین اگر یک مقدار نمادین از زمینه^۸ برنامه خارج شود، مثلاً 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 (مثلاً کد آن در دسترس نباشد)، پس تلاش بیشتری لازم است تا بتوان آن کتابخانه را نیز به صورت نمادین اجرا کرد.
- 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 برنامه 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 برنامه می‌شود.

یک مثال پرکاربرد از این نوع می‌تواند Intentها باشد که سیستم پیام‌رسانی بین مولفه‌های 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 سیستمی شده و بعد از آن وارد مولفه مقصد می‌شود.

۳. برنامه‌های اندروید رخدادمحور^۹ هستند. به این معنی که در اجرای نمادین، موتور اجرا باید منتظر کاربر بماند تا با تعامل با برنامه 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.

⁷ Path divergence

⁸ Context

⁹ Component

¹⁰ Event Driven

پیش از این پژوهش کارهایی در حوزه آزمون برنامه‌های اندرویدی با سه رویکرد متفاوت انجام شده است. در رویکرد اول به طور بی‌قاعده^{۱۱} ورودی برنامه تولید می‌گردد. در ابزار Monkey [۳] به صورت دلخواه سعی می‌شود تا ورودی‌های آزمون برای برنامه تولید شود. مشکل اصلی این روش آن است که پوشش مناسبی از مسیرهای مختلف برنامه را نمی‌توان داشت. در رویکرد دوم ورودی برنامه به طور نظام‌مند^{۱۲} تولید می‌گردد. در ابزار Sig-Droid [۴] با استفاده از یک روش مشخص مانند اجرای نمادین به صورت جعبه‌سفید^{۱۳} سعی می‌شود تا ورودی‌های برنامه تولید گردد. Sig-Droid تمام مسیرهای موجود در برنامه را به صورت «نمادین» اجرا می‌کند و همان طور که نویسنده بیان کرده است هدف آن پوشش هرچه بیشتر این مسیرها است. در رویکرد سوم مانند ابزار Swifthand [۵] مدلی از برنامه مانند مدل رابط گرافیکی کاربر^{۱۴} از برنامه استخراج می‌گردد و سپس بر اساس این مدل‌ها ورودی‌هایی برای برنامه تولید می‌شود که مسیرهای ناشناخته برنامه را بییماید.

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 [۶] 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^{۱۵} وجود نداشته باشد. همچنین، می‌توان با روش‌های ضد ایستا^{۱۶} نیز مقابله کرد، که در این روش‌ها به منظور جلوگیری از تحلیل ایستای برنامه از یک سری تکنیک‌ها مثل بارگذاری پویای کد^{۱۷} استفاده می‌شود. چالش‌هایی که در این روش وجود دارد عبارتند از:

¹¹ Random

¹² Systematic

¹³ White Box

¹⁴ GUI

¹⁵ False Positive

¹⁶ Anti-Static

¹⁷ Dynamic Code Loading

- **انفجار مسیر:** در برنامه‌های دنیای واقعی تعداد خطوط برنامه 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:** برنامه‌های اندرویدی در یک چارچوب کاری خاص به خود اجرا و تولید می‌شوند. همان طور که پیش از این نیز گفته شد، برنامه‌های اندرویدی فراخوانی‌های زیادی به SDK دارند. همچنین این برنامه‌ها رخدادهای محور هستند. در اجرای پویا-نمادین باید رخدادهای مختلف از جمله رخدادهای مرتبط با تعامل کاربر با برنامه نیز مدل‌سازی شوند تا فرایند تحلیل و آزمون شبیه به اجرای واقعی برنامه باشد.

در ادامه کارهایی را شرح خواهیم داد که در حوزه آزمون و یا آزمون امنیتی برنامه‌های اندرویدی با اجرای پویا-نمادین صورت گرفته است. برای اولین بار ACTEVE [۷] از اجرای پویا-نمادین برای آزمون برنامه‌های اندرویدی استفاده کرده است. در این کار با تغییر دادن SDK سعی شده است تا رخدادهای لمس صفحه^{۱۸} به شکل نمادین تولید شود. علاوه بر آن با تولید جایگشت‌های مختلف از رخدادهای 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 رخدادهای و اجرای آنها باعث می‌شود که فرایند آزمون با انفجار مسیر روبه‌رو شود. در [۸] از ابزار پیشین با عنوان Condroid برای تشخیص وجود دزافزار^{۱۹}ها استفاده شده است که همان اشکالات ACTEVE را به ارث برده است. در AppIntent [۹] از اجرای پویا-نمادین برای کشف نقض حریم خصوصی در برنامه‌های 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 برنامه‌های اندرویدی ارائه شده است. این ابزار نزدیک‌ترین کار موجود به کار ما است. در این ابزار سعی شده است برنامه‌ها روی JVM کامپایل شوند تا بتوان به کمک موتورهای اجرای نمادین جاوا، برنامه را آزمود. این ابزار تمام مسیرهای موجود در برنامه را به صورت نمادین اجرا می‌کند.

¹⁸ Tap Event

¹⁹ Malware

بر اساس آخرین دانسته ما تاکنون [۱۰]، ابزاری برای تشخیص آسیب‌پذیری در برنامه‌های اندرویدی با اجرای پویا-نمادین ارائه نشده است. کارهایی که تا به حال در مورد تشخیص آسیب‌پذیری در برنامه‌های اندرویدی انجام شده است، همگی از تحلیل ایستای کد برنامه استفاده می‌کنند. در نوشته‌جات مربوطه هم اگر نامی از تحلیل پویا آمده است منظور اجرای پویا-نمادین و یا نمادین برنامه نیست. بلکه منظور اجرای برنامه در محیط شبیه‌ساز است که به این وسیله درستی تشخیص خود را مورد آزمایش قرار می‌دهند. از جمله ابزارها و پژوهش‌های گذشته مربوط به آزمون امنیتی برنامه‌های اندرویدی که از اجرای پویا-نمادین و یا نمادین استفاده نمی‌کنند، می‌توان به TaintDroid [۱۱]، ScanDroid [۱۴]، CHEX [۱۳]، APSET [۱۴]، VulHunter [۱۵] و کارهای [۱۶] و [۱۷] اشاره کرد.

در منبع [۱۸] عنوان شده است که آسیب‌پذیری تزریق می‌تواند موجب نشت اطلاعات حساس کاربر شود. آسیب‌پذیری تزریق در اندروید می‌تواند تزریق دستور به پوسته^{۲۰} سیستم عامل، تزریق دستورات SQL به پایگاه داده 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:

۱. بهبود اجرای پویا-نمادین برنامه‌های اندرویدی برای تولید خودکار ورودی آزمون

۲. اجرای پویا-نمادین برای تشخیص آسیب‌پذیری تزریق در برنامه‌های اندرویدی

لازم به ذکر است که بهبود اجرای پویا-نمادین و ارائه یک روش هیوریستیک برای این منظور جزئی از پیشنهاد پروژه اولیه نبوده است. این موضوع به صورت جداگانه طرح و تعریف شده است.

برای بهبود اجرای پویا-نمادین (سوال اول)، ابتدا ما apk برنامه را دیکامپایل می‌کنیم. سپس برای اینکه بتوانیم بر روی JVM برنامه را با موتور^{۲۱} SPF [۱۹] به شکل پویا-نمادین اجرا کنیم، لازم است تا تابع نقطه شروع به برنامه را تولید نماییم. برای این منظور ما از ایده تحلیل ایستا استفاده کردیم [۲]. ولی در

²⁰ Shell

²¹ Symbolic Path Finder

تحلیل ایستا بر خلاف [۲] از نوآوری «گراف فراخوانی توابع»^{۲۲} در اندروید بهره بردیم. همچنین، برای اینکه تحلیل هدفمند شود ما ایده پیمایش روبه‌عقب گراف را پیشنهاد کرده‌ایم.

در این پژوهش ما می‌خواهیم که اجرای پویا-نمادین برنامه به صورت هدفمند صورت گیرد تا بتوان خطاهای برنامه را سریع‌تر یافت. برای این هدف، ما ایده ترکیب تحلیل ایستا و پویا را مطرح کردیم. در تحلیل ایستا ما برای اولین بار از پیمایش رو به عقب «گراف کنترل جریان بین تابعی»^{۲۳} بهره می‌بریم. در پیمایش رو به عقب، از عبارت^{۲۴} رخداد خطا تا عبارت ریشه را می‌پیماییم و «پشته شاخه‌های اولویت‌دار» را مبتنی بر آن تولید می‌کنیم. برای حل مسئله رخدادمحور بودن و کامپایل شدن در JVM از کلاس‌های Mock به جای کلاس‌های SDK استفاده کردیم [۴]. 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 سربر کلاس‌های اصلی را داشته باشد. ایده کلاس‌های Mock از ایده استفاده از «خلاصه تابع»^{۲۵} [۲۰] به جای تابع اصلی در برنامه گرفته شده است.

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 روش هیوریستیک^{۲۶} خود را ارائه می‌دهیم که باعث می‌شود، به جای پیمایش عمق‌اول^{۲۷} در اجرای پویا-نمادین، در هر دستور شرطی متناسب با اطلاعات پشته، شاخه بهینه را اجرا کنیم که به خطا خواهد رسید. این دو مورد موجب کاهش قابل توجه هزینه زمانی و سربر اجرای برنامه می‌شود. در نهایت با استفاده از ابزار Robolectric [۲۱] و ورودی‌های آزمون استخراج شده از اجرای پویا-نمادین، برنامه را اجرا می‌نماییم تا صحت عملکرد روش هیوریستیک و ورودی‌های تولید شده را بررسی کنیم.

²² Call Graph

²³ Inter-Control Flow Graph (ICFG)

²⁴ Statement

²⁵ Function Summary

²⁶ Heuristic

²⁷ DFS

برای ارزیابی راه کار ارائه شده، علاوه بر ۱۰ برنامه‌ای که خودمان مطرح و پیاده‌سازی کردیم، ۴ برنامه از برنامه‌های مورد آزمون در ابزار Sig-Droid را آزمودیم. این برنامه‌ها از مخزن F-Droid [۲۲] انتخاب شده‌اند که مخزن متن‌باز^{۲۸} برنامه‌های اندرویدی هستند. نتایج نشان می‌دهد که روش هیوریستیک ما باعث می‌شود با سرعت بیشتر و پوشش کمتری نسبت به ابزار Sig-Droid به خطاهای موجود در برنامه برسیم.

برای اجرای پویا-نمادین به منظور تشخیص آسیب‌پذیری تزریق (سوال دوم)، باز از موتور SPF استفاده کرده‌ایم. ابتدا تابع ورودی به برنامه را تولید کردیم. با استفاده از ایده خودمان و کمک گرفتن از تحلیل ایستا و استخراج گراف فراخوانی توابع، می‌توان این تابع را داشت. برای کاهش هزینه اجرای ابزار ما ایده پیمایش روبه‌عقب گراف را ارائه کردیم. در این ایده ما گراف را از تابع آسیب‌پذیر^{۲۹} (مثلا query) تا تابع منبع^{۳۰} (مثلا EditText) به صورت برعکس پیمایش کردیم و تابع نقطه شروع به برنامه را بر این تولید کردیم. برای حل مسئله رخدادمحور بودن و کامپایل شدن در JVM از کلاس‌های Mock به جای کلاس‌های SDK استفاده کردیم [۴]. برای اینکه هزینه اجرای ابزار را باز هم کمتر کنیم، با توجه به مسیری که از گراف فراخوانی توابع بدست آوردیم، ایده استفاده از کلاس‌های Mock به جای کلاس‌هایی از برنامه که نیاز به اجرا ندارند را مطرح کردیم.

ما برای اولین بار روشی برای تشخیص آسیب‌پذیری تزریق SQL در برنامه‌های اندرویدی ارائه کردیم. برای تشخیص این آسیب‌پذیری، ما مولفه‌ای به SPF اضافه کردیم. در این مولفه ما ایده اضافه کردن تحلیل آرایش^{۳۱} به اجرای پویا-نمادین را مطرح کردیم. برای اینکه بتوانیم گردش داده آرایش شده^{۳۲} در برنامه را به درستی انجام دهیم، ایده کلاس‌های «Mock نمادین» برای کلاس پایگاه‌داده SQLite و کلاس‌های منبع برنامه (مثلا EditText) را برای اولین بار عنوان کردیم. این کلاس‌ها شامل همان توابع و کلاس‌های اصلی هستند با این تفاوت که بدنه آنها حذف شده و خروجی توابع آنها نمادین خواهد

²⁸ Open Source

²⁹ Sink Method

³⁰ Source Method

³¹ Taint Analysis

³² Tainted Data

بود. خروجی تحلیل SPF شامل دنباله پشته^{۳۳} برنامه تا تابع آسیب‌پذیر، شناسه تابع منبع، تصفیه‌شدن^{۳۴} یا نشدن داده ورودی توسط برنامه‌نویس و شناسه تابع نشت^{۳۵} داده‌ها (مثلا TextView) است. در نهایت با استفاده از این خروجی‌ها و تابع ورودی بدست آمده از تحلیل ایستا با ابزار Robolectric قابلیت بهره‌جو^{۳۶}یی برنامه به تزریق SQL را بررسی کردیم.

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

به طور مختصر در این پژوهش نوآوری‌های زیر صورت گرفته است:

- اجرای برنامه روی JVM با استفاده از نقطه ورودی به دست آمده از تحلیل ایستا روی گراف فراخوانی توابع.
- اجرای روش هیوریستیک پیشنهادی در SPF با تحلیل ایستا روی گراف کنترل جریان بین تابعی و به دست آوردن پشته شاخه‌های اولویت‌دار.
- اجرای پویا-نمادین برای برنامه‌های اندروید با استفاده از ایده کلاس‌های Mock نمادین و جلوگیری از مشکل انحراف مسیر و رخدادمحور بودن با استفاده از ایده ساخت Mock.
- ارائه یک روش هیوریستیک با ایده ترکیب تحلیل ایستا و پویا که موجب می‌شود تا اجرای شاخه‌های دارای خطا را اولویت دهیم.
- اضافه کردن تحلیل آرایش به اجرای پویا-نمادین به منظور تشخیص آسیب‌پذیری تزریق و اضافه کردن یک مولفه به SPF نسبت به Sig-Droid برای این منظور در اندروید.
- محدود کردن فرایند تحلیل به تابع‌های نقطه شروعی که منجر به بروز آسیب‌پذیری می‌شوند و محدود کردن 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 برنامه.

³³ Stack Trace

³⁴ Sanitize

³⁵ Leackege Method

³⁶ Exploitability

- استفاده از ایده Mock نمادین برای کلاس و تابع‌های آسیب‌پذیر به منظور گردش داده‌های آرایش‌شده در برنامه 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 برنامه‌های اندرویدی، کارها و ابزارهای ارائه شده برای تشخیص آنها را معرفی خواهیم کرد. در پایان فصل هم آسیب‌پذیری تزریق SQL در برنامه‌های 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 نهایت در فصل ششم، منابع و مراجع 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 برنامه‌های 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-

```

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: }

```

نمادین در این است که در اجرای پویا-نمادین علاوه بر اجرای نمادین به صورت هم‌زمان برنامه به شکل عینی نیز اجرا خواهد شد.

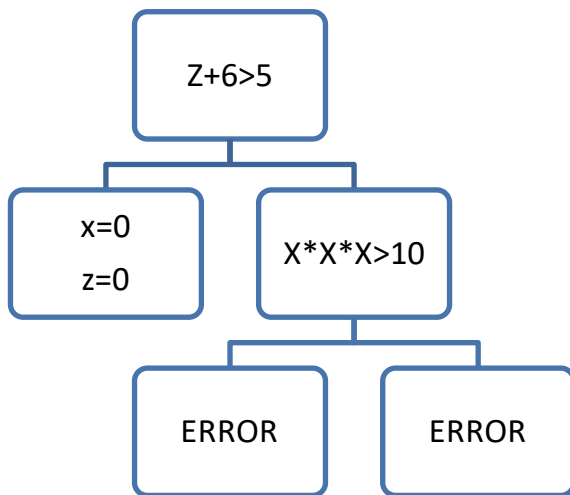
ابتدا اجرای نمادین برنامه شکل ۲-۱ را بررسی می‌کنیم. وقتی در اجرای نمادین متغیری نمادین در نظر گرفته می‌شود (خط ۱۱ و ۱۲) به این معنی است که آن متغیر نماینده تمام مقادیر ممکن برای آن نوع است. مثلاً متغیر x نماینده تمام مقادیر ممکن برای نوع int است. این مقدار را با حرف بزرگ نشان خواهیم داد. مثلاً مقدار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

شکل ۲-۱ نمونه برنامه ساده

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 مواجهه با دستورات شرطی از قبیل `if`، شرط مربوط به آن را به صورت یک عبارت منطقی به عنوان شرط مسیر^{۳۷} 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^{۳۸} شرط‌های دستوره‌های شرطی موجود در آن مسیر بدست می‌آید. شرط مسیر در ابتدا مقدار درست^{۳۹} 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^{۴۰}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 «حل‌کننده قید»^{۴۱} 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. مثلاً به ازای $x=0$ و $y=0$

³⁷ Path Condition

³⁸ And

³⁹ True

⁴⁰ Execution Tree

⁴¹ Constraint Solver

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

در روش پویا-نمادین متغیرها علاوه بر شکل نمادین به صورت مقدار عینی^{۴۲} 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. همان‌طور که گفته شد، حل‌کننده قید توانایی حل عبارت‌های غیرخطی 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. مثلاً $x=3$ 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 ساختمان داده‌های 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. مثلاً برنامه‌های 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

⁴² 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 روش 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.
- **حل کننده‌های قید:** حل کننده‌های 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 آزمون 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 که از ترکیب این دو روش استفاده شود تا مزیت هر کدام 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.

۴-۲ 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 ۱-۲ کارهای گذشته شاخص آمده‌اند.

جدول ۱-۲ کارهای گذشته

ایده	نوع	پلتفرم	زبان برنامه مورد آزمون	ابزار	سال	
عمق اول	آفلاین	کامپیوتر شخصی	C	DART	۲۰۰۵	۱
مدل سازی حافظه، عمق اول کراندار، بهینه سازی	آفلاین	کامپیوتر شخصی	C	CUTE	۲۰۰۵	۲
ایده‌های ابزار CUTE، همروندی	آفلاین	کامپیوتر شخصی	جاوا	jCUTE	۲۰۰۶	۳
مدل سازی حافظه، عمق اول و سطح اول ترکیبی، بهینه سازی، حل کننده قید: STP	آنلاین	کامپیوتر شخصی	C	EXE	۲۰۰۶	۴
CUTE، ترکیب اجرای دلخواه و پویا-نمادین	آفلاین	کامپیوتر شخصی	C	Hybrid	۲۰۰۷	۵
DART، ترکیب Compositional و پویا-	آفلاین	کامپیوتر شخصی	C	Compositional	۲۰۰۷	۶

نمادین						
EXE، بهینه‌سازی انتخاب مسیر و حل‌کننده قید، متن‌باز	آنلاین	کامپیوتر شخصی	C	KLEE	۲۰۰۸	۷
متن‌باز، بهینه‌سازی کارهای گذشته	آفلاین	کامپیوتر شخصی	جاوا	jFUZZ	۲۰۰۹	۸
ترکیب واریسی مدل و اجرای نمادین	آفلاین	کامپیوتر شخصی	جاوا	SPF	۲۰۱۰	۹
متن‌باز، حل‌کننده قید: SMT، معماری برنامه نویسی سوکت	آفلاین	کامپیوتر شخصی	جاوا	LCT	۲۰۱۱	۱۰
آسیب‌پذیری سرریز بافر، خاصیت ایمنی	آفلاین	کامپیوتر شخصی	باینری و C	AEG	۲۰۱۱	۱۱
متن‌باز، رخدادمحور	آفلاین	Phone	اندروید	ACTEVE	۲۰۱۲	۱۲
آسیب‌پذیری سرریز بافر، قالب رشته، مدل‌سازی حافظه	ترکیبی	کامپیوتر شخصی	باینری	MAYHEM	۲۰۱۲	۱۳
ثبت-بازاجرای انتخابی، مقادیر سایه	آفلاین	وب	جاوا اسکریپت	Jalangi	۲۰۱۳	۱۴
کشف نشت حریم خصوصی	آفلاین	Phone	اندروید	AppIntent	۲۰۱۳	۱۵
استفاده از کلاس‌های Mock، گراف فراخوانی توابع	آفلاین	Phone	اندروید	SIG-Droid	۲۰۱۵	۱۶
پویا-نمادین + Call Flow، Graph، تشخیص دژافزار	آفلاین	Phone	اندروید	Condroid	۲۰۱۵	۱۷
ترکیب -instrumented	ترکیبی	کامپیوتر شخصی	باینری	Driller	۲۰۱۶	۱۸

Genetic- فازر با پویا-نمادین		شخصی				
تحلیل باینری، ابزار Valgrind، تشخیص دژافزار	آفلاین	Phone	اندروید- باینری	Malton	۲۰۱۷	۱۹
اجرای نمادین برای تشخیص آسیب پذیری در چارچوبه کاری اندروید و نه در برنامه‌های اندرویدی	آفلاین	Phone	اندروید	System Service Call-oriented...	۲۰۱۷	۲۰

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 Ip 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 حل‌کننده قید ندارد همچنین این ابزار در حل قیدهای مربوط به اشاره‌گرها مشکل دارد.
۲. در سال ۲۰۰۵، ابزار CUTE [۲۳] با روش پویا-نمادین آفلاین ارائه شد که از Ipsolve 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 حل‌کننده قید دارد. روش‌های بهینه‌سازی آن عبارتند از: بررسی سریع ارضاناپذیری، حذف قیدهای معمول و حل افزایشی.
۳. در سال ۲۰۰۶، ابزار JCUTE [۲۴] با روش پویا-نمادین آفلاین، ارائه شد که از Ipsolve 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 حل‌کننده قید دارد.

۴. در سال ۲۰۰۶، ابزار EXE [۲۵] با روش پویا-نمادین آنلاین، ارائه 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 بر آن بهینه‌سازی برای ارسال قیده‌ها به حل‌کننده قید دارد. ایده‌های این ابزار در این مورد استفاده از روش Cache و شناسایی زیرقیده‌های مستقل و حذف زیرقیده‌های بی‌ارتباط است.

۵. اجرای هیبرید [۲۶] به صورت ترکیبی اجرای دلخواه^{۴۳} 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 می‌یابد تا بتواند به صورت عمق‌محدود به پوشش بیشتری از کد برسد. دوباره بعد از یافتن مسیر جدید اجرا به عینی تغییر می‌یابد. 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. از نظر نویسندگان پوشش کامل نشانه‌ای برای قابل اعتماد بودن^{۴۴} کد می‌تواند باشد یا نباشد. این به تحلیل و هدف آن بر می‌گردد.

۶. کار مورد شش در جدول [۲۰]، از DART به عنوان موتور پویا-نمادین استفاده می‌کند. هدف این کار توسعه DART برای برنامه‌های واقعی با تعداد خط کد بالاست به همین دلیل از تحلیل ایستای Compositional استفاده می‌کند که برای توابع function summary استخراج می‌کند و به جای اجرای هر باره یک تابع از summary 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 حل‌کننده قید 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

⁴³ Random

⁴⁴ Reliability

دلخواه و انتخاب برای پوشش بیشترین مسیرها را به صورت ترکیبی استفاده می‌کند. بر اساس یک سری روش هیوریستیک در روش اول، به حالت‌ها وزن اختصاص داده می‌شود و سپس به صورت دلخواه یکی از این حالت‌ها انتخاب می‌شوند. در روش دوم، هیوریستیک‌ها بر اساس کمترین فاصله تا دستور پوشش داده نشده، بیشینه فراخوانی حالت و یا اینکه یک حالت اخیراً 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 به حل‌کننده قید از روش‌هایی مثل روش Cache استفاده می‌کند. این ابزار گسترش یافته ابزار EXE است.

۸. jFuzz [۴۴] 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 [۴۵] پیاده سازی شده است.

۹. برای اجرای نمادین برنامه‌های به زبان جاوا، ابزار SPF^{45} [۱۹] 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 حل‌کننده‌های 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، ما با گسترش SPF به دنبال تشخیص آسیب‌پذیری تزریق به برنامه‌های اندرویدی هستیم.

۱۰. ابزار LCT [۲۸] ابزار متن باز روی جاوا است. در این ابزار سعی شده از معماری سرویس دهنده-سرویس گیرنده^{۴۶} برای ارتباط بین حل‌کننده قید و تحلیلگر استفاده کند. مشکل این ابزار این

⁴⁵ Symbolic Path Finder

⁴⁶ client-server

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 اطلاعاتی از مقادیر و چینش داده‌ها در زمان اجرا ندارد. در مقابل تحلیل باینری مقیاس‌پذیر نیست و مفاهیمی مثل متغیرها ساختمان داده‌ها (آرایه‌ها و غیره) در آن معنی ندارد. تنها با فریم‌های پشته و دستورات پرش و آدرس‌های حافظه سر و کار دارد. در AEG [۲۹] 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 حل‌کننده قید داده می‌شود تا ورودی مناسب تولید شود.
- سپس به صورت پویا و با استفاده از ورودی تولید شده، فایل باینری برنامه تحلیل می‌شود تا اطلاعات زمان اجرا^{۴۷} یعنی ساختار حافظه مثل آدرس بافر سرریز شده و آدرس بازگشت استخراج شود.
- AEG قیده‌های جدیدی مربوط به اطلاعات ساختار حافظه تولید می‌کند و به شرط مسیر اضافه می‌کند. این قیده‌ها باید شامل shell code و آدرس بازگشت به shell code باشند. سپس شرط مسیر به حل‌کننده قید داده می‌شود تا ورودی مناسب تولید شود.
- در نهایت AEG ورودی تولید شده را به برنامه می‌دهد تا بررسی کند که کد بهره‌بردار^{۴۸} آیا اجرا می‌شود یا نه! اگر حل‌کننده 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 برنامه‌های اندرویدی^{۴۹} ارائه شده است. این برنامه‌ها رخدادمحور^{۵۰} هستند. منظور از رخدادمحور بودن این است که کاربر با برنامه تعامل

⁴⁷ Runtime

⁴⁸ Exploit

⁴⁹ Android Apps

⁵⁰ Event Driven

دارد و رفتار او در فرایند اجرای برنامه موثر است. یعنی علاوه بر داده‌ها، رخدادهای هم مسیر اجرای برنامه را تعیین می‌کنند. چالش این آزمون تولید یک رخداد و همچنین تولید ترتیبی از رخدادهای است.

در این ابزار از روش پویا-نمادین برای تولید رخدادهای استفاده می‌شود. برای این منظور ⁵¹ SDK و برنامه تحت آزمون باید تجهیز ⁵² شوند. سپس در حین اجرای یک رخداد عینی، یک رخداد به صورت نمادین هم تولید می‌شود که تمام قیده‌های مسیر را در خود نگهداری می‌کند. با این روش برای ترتیبی از رخدادهای باید همه حالت‌های وقوع رخدادهای بررسی شود. (دوتایی، سه‌تایی، چهارتایی و غیره) و جای‌گشت‌های مختلف رخدادهای در هر ترتیب نیز در نظر گرفته شود که فضای حالت خیلی بزرگی دارد. این کار محدود به رخداد ضربه ⁵³ است. علاوه بر آن همان‌طور که گفته شد، نیاز به بهینه‌سازی برای کاهش فضای حالت در ترتیب‌های مختلف از رخدادهای دارد. در این کار با حذف ویجت‌های 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 [۳۰] 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 بدانند چه کدهایی در اختیار مهاجم است.

⁵¹ Software Development Kit

⁵² Instrument

⁵³ Tap Event

⁵⁴ Widget

- واحد CEC⁵⁵ برنامه آسیب‌پذیر را دریافت می‌کند. به SES⁵⁶ وصل می‌شود تا مقاردهی‌های اولیه صورت پذیرد. سپس کد به صورت عددی اجرا می‌شود و همزمان تحلیل آرایش⁵⁷ 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 نمی‌شود و شاخه آلوده به SES برای اجرای نمادین ارسال می‌شود. SES مشخص می‌کند که آیا اجرای شاخه ممکن هست یا نه!
 - واحد SES به صورت موازی با CEC اجرا می‌شود و بلاک‌های کد را دریافت می‌کند. این بلاک‌ها به زبان میانی تبدیل می‌شوند و به صورت پویا-نمادین اجرا می‌شود. مقادیر عددی مورد نیاز از CEC دریافت می‌شود.
 - فرمول قابلیت اکسپلویت مشخص می‌کند که:
 - آیا مهاجم می‌تواند کنترل اجرای دستورات یا
 - اجرای PAYLOAD را بدست آورد یا نه؟
 - وقتی به یک پرش آلوده می‌رسد SES تصمیم می‌گیرد که آیا FORK لازم هست یا نه. اگر باشد اجراهای جدید اولویت بندی شده و یکی اجرا می‌شود. اگر منابع تمام شوند SES رویه بازگشت را اجرا می‌کند. در نهایت بعد از اتمام اجرای یک پردازش تعدادی موردآزمون تولید می‌شوند.
 - در پرش‌های آلوده یک فرمول بهره‌جو⁵⁸ تولید و به SES داده می‌شود اگر قابل ارضا بود یعنی کد از این مسیر آسیب‌پذیر است.
۱۴. ابزار Jalangi [۳۱] 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 انتخابی⁵⁹ استفاده

⁵⁵ Concrete Execution Client

⁵⁶ Symbolic Execution Server

⁵⁷ Taint Analysis

⁵⁸ Exploit

⁵⁹ Selective Record-Replay

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^{۶۰} استفاده می‌کند. این مقادیر اطلاعاتی اضافی (مثل آرایش شدن یا نمایش نمادین) را در مورد داده‌های اصلی در خود نگهداری می‌کنند. از این مقادیر در اجرای نمادین یا تحلیل آرایش 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 برنامه‌های اندرویدی و بدون از دست رفتن پوشش کد بالا است. در ابزار AppIntent از تحلیل آرایش ایستا استفاده شده است که با استفاده از آن تمامی انتقال داده‌های حساس و دنباله رویدادهای مربوط به آنها استخراج می‌شود. در ادامه با اجرای نمادین هدایت‌شده توسط اطلاعات به دست آمده از تحلیل آرایش 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 برنامه‌های اندرویدی ارائه شده است. در این ابزار سعی شده است برنامه‌ها روی JVM^{۶۱} کامپایل شوند تا بتوان به کمک موتورهای اجرای نمادین جاوا، برنامه را آزمود. این ابزار تمام مسیرهای موجود در برنامه 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^{۶۲} 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.

⁶⁰ Shadow Values

⁶¹ Java Virtual Machine

⁶² Main Method

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، مثلاً تعداد زیاد دستورات شرطی پشت سرهم، سعی می‌کند بمب منطقی را در برنامه‌های اندرویدی تشخیص دهد. این ابزار همان مشکلات ACTEVE یعنی انفجار مسیر را به ارث برده است.

۱۸. ابزار Driller [۳۲] 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.
- 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 به فازر سپرده می‌شود و اجرا ادامه پیدا 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 خود را به موتور پویا-نمادین می‌دهد و اجرای پویا-نمادین تنها سعی می‌کند با مکمل کردن یکی از شرطهای مسیر ورودی از فازر به مسیری جدیدی برسد.

۱۹. ابزار Malton [۳۳] در سال ۲۰۱۷ برای تشخیص دژافزار در برنامه‌های اندرویدی ارائه شده است. در این ابزار سعی شده است با کمک گرفتن از ابزار Valgrind [۳۴] تحلیل برنامه‌های اندرویدی به صورت باینری صورت پذیرد. در این ابزار برنامه در موبایل اجرا می‌شود. سپس اطلاعات مورد نیاز تحلیل به کامپیوتر منتقل می‌شود. به این وسیله مسئله وابستگی به SDK سعی شده است که حل شود.

۲۰. ابزار [۳۵] به منظور تشخیص آسیب‌پذیری در چارچوبه کاری اندروید ارائه شده است. در این کار تلاش می‌شود با استفاده از اجرای نمادین آسیب‌پذیری‌های موجود در SDK پیدا شوند.

نویسندگان مقاله معتقدند که وجود چنین آسیب‌پذیری‌هایی می‌تواند امنیت و حریم خصوصی تعداد زیادی از کاربران را به مخاطره بیاندازد.

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 تاکنون بررسی شدند. همان طور که در بخش ۲-۲ نیز گفته شد، اجرای پویا-نمادین با تعدادی چالش روبه‌رو است. در نوشته‌جات گذشته سعی شده است برای هر یک از این چالش‌ها راه حلی ارائه شود. اما همان طور که دیده می‌شود، نمی‌توان گفت که راه حلی جامع و کلی برای رفع هر کدام از آنها ارائه شده است و هر یک از کارهای گفته شده دارای اشکالات و نقص‌هایی هستند. بنابراین می‌توان گفت که در مورد هر کدام از چالش‌های عنوان شده می‌توان سوال پژوهشی تعریف کرد و به آن پاسخ داد. علاوه بر سوالات مطرح در مورد خود اجرای پویا-نمادین و چالش‌های مرتبط با آن، سوالاتی در مورد کاربرد این روش در حوزه‌های دیگر مثل کشف آسیب‌پذیری، کشف و تحلیل دزافزار، کشف نقض حریم خصوصی و غیره قابل تعریف هستند.

همان طور که دیده می‌شود، اجرای پویا-نمادین روشی است که می‌تواند به پوشش بالایی از کد برسد. همچنین چون برنامه اجرا می‌شود، مثبت نادرست ندارد. پس می‌تواند روشی مناسب برای تحلیل نرم‌افزارها برای تشخیص آسیب‌پذیری در آنها باشد. ما از این روش در تحلیل برنامه‌های اندرویدی برای تشخیص آسیب‌پذیری استفاده کرده‌ایم. همچنین با بهره‌گیری از تحلیل ایستا و ترکیب آن با اجرای پویا-نمادین توانستیم با مشکل انفجار مسیر مقابله کنیم.

۳

فصل سوم تشخیص آسیب پذیری

تشخیص آسیب پذیری

یکی از مشکلاتی که در امنیت نرم افزارها مطرح است مسئله آسیب پذیری های موجود در نرم افزارهاست. نحوه کشف این آسیب پذیری ها، از جمله مطالب مورد علاقه محققان بوده است. مطالبی که در قسمت ۳-۱ مورد بحث قرار می گیرند مطالعه ای بر روی آسیب پذیری های نرم افزاری و نحوه کشف آنها به همراه دسته بندی روش های تشخیص است. همچنین در هر مورد تعدادی ابزار معرفی شده است. پس از آن به بررسی یکی از آسیب پذیری های معروف یعنی آسیب پذیری تزریق خواهیم پرداخت. در ادامه آسیب پذیری های مطرح در برنامه های اندرویدی و روش های تشخیص آنها را بررسی می کنیم و در آخر به طور خاص آسیب پذیری تزریق SQL در برنامه های اندرویدی را به تفصیل بررسی خواهیم کرد. در این پژوهش ما به دنبال تشخیص آسیب پذیری تزریق در برنامه های اندرویدی هستیم. از میان آسیب پذیری های تزریق متنوع موجود از جمله تزریق دستور به پوسته سیستم عامل، تزریق Intent، تزریق کد جاوا اسکریپت به WebView و تزریق SQL، ما تزریق SQL را به دلیل متداول بودن و درک بهتر انتخاب کرده ایم. لازم به ذکر است که روشی که ما برای تشخیص آسیب پذیری تزریق SQL ارائه کرده ایم، برای همه آسیب پذیری های تزریق گفته شده در بالا شباهت دارد. جزئیات این روش به تفصیل در بخش ۴-۲ آمده است. شباهت همه این آسیب پذیری ها در این است که همه از طریق ورودی آلاینش شده^{۶۳} به برنامه اتفاق می افتند. این ورودی اگر به یک تابع آسیب پذیر وارد شود و سپس نتیجه پردازش از طریق تابعی به بیرون نشت پیدا کند، آسیب پذیری اتفاق می افتد. در همه این آسیب پذیری ها این روند وجود دارد. تفاوتی که در آنها دیده می شود در این است که توابع آسیب پذیر و قاعده ای که به واسطه آن گفته می شود آسیب پذیری اتفاق افتاده، متفاوتند. با مطالعه هر کدام از این آسیب پذیری ها می توان این تفاوت ها را پیدا کرد و به روشی که ما ارائه کرده ایم اضافه کرد.

۳-۱ مطالعه ای بر روش های تشخیص آسیب پذیری در نرم افزارها

در مقاله [۳۶] دو تعریف در رابطه با تشخیص آسیب پذیری آمده است:

⁶³ Tainted Input

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 توصیف^{۶۴}، توسعه یا تنظیمات نرم‌افزار که اجرای آن توسط مهاجم به طور صریح یا ضمنی، یکی از خط مشی‌های امنیتی را نقض می‌کند.

هر دو تعریف بیان شده، شامل نقصی در فرایند تولید نرم افزار هستند که مهاجم می‌تواند از آن سوء استفاده کند. در این پژوهش، مورد نظر ما آسیب‌پذیری‌های نرم‌افزاری است و این تعریف از آسیب‌پذیری با هر دو تعریف بالا مطابقت دارد، ولی به تعریف اول بیشتر نزدیک است. چون که برنامه‌نویس به دلیل عدم آگاهی یا به صورت سهوی برنامه خود را طوری پیاده‌سازی می‌کند که موجب ایجاد آسیب‌پذیری و سوء استفاده‌های بعدی می‌شود. در تحلیل ما کد منبع برنامه اندرویدی مورد نظر است، نه طراحی و یا توصیف آن.

تحقیقات در حوزه آسیب‌پذیری به دو دسته طبقه‌بندی می‌شود:

۱. تحلیل آسیب‌پذیری: در این حوزه بر روی ویژگی‌های آسیب‌پذیری‌های یافته شده پیشین تحقیقات انجام می‌شود. مثل دلیل، محل وقوع آسیب‌پذیری، ویژگی‌های پیاده‌سازی برنامه که موجب آسیب‌پذیری می‌شوند و یا روند تشخیص چنین آسیب‌پذیری‌هایی بررسی می‌شوند. هدف از تحقیقات در این حوزه، ارائه روند شناسایی آسیب‌پذیری‌های مشابه در برنامه‌های دیگر، راه‌های جلوگیری و مقابله با آنها و همچنین شناسایی آسیب‌پذیری‌های جدید و شناخته نشده است.
۲. کشف آسیب‌پذیری: هدف از پژوهش در این حوزه، ارائه و بررسی روش‌های مختلف برای 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^{۶۵}، تحلیل پویا^{۶۶}، روش فازینگ^{۶۷} و آزمون نفوذ^{۶۸} در نرم‌افزارها تقسیم می‌شود.

⁶⁴ Specification

⁶⁵ Static Analysis

۳-۱-۱ تحلیل ایستا

در مقاله [۳۶] تحلیل ایستا به عنوان روشی برای تشخیص آسیب پذیری به صورت زیر تعریف شده است:

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.

همچنین در این مقاله بیان شده است که بر اساس تئوری رایس^{۶۹} هر مسئله غیر بدیهی در مورد برنامه‌ها را می‌توان به مسئله توقف^{۷۰} کاهش داد. پس وجود آسیب پذیری در یک برنامه تصمیم ناپذیر^{۷۱} است و بعضی از آسیب پذیری‌ها توسط روش ایستا اساساً 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^{۷۲} و منفی نادرست^{۷۳} 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^{۷۴} است. با این ابزار رشته‌های موجود در کد برنامه بررسی شده و با لیستی از رشته‌هایی از آسیب پذیری‌ها مطابقت داده می‌شود. ابزار دیگر IST4 و RAT است. این دو ابزار از تحلیل کلمات^{۷۵} 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 آسیب پذیری‌ها مطابقت داده می‌شود. ابزار دیگر SWORD4J است. این

⁶⁶ Dynamic Analysis

⁶⁷ Fuzzing

⁶⁸ Penetration Testing

⁶⁹ Rice Theory

⁷⁰ Halting Problem

⁷¹ Undesirable

⁷² False Positive

⁷³ False Negative

⁷⁴ Unix

⁷⁵ Lexical Analysis

ابزار با پیاده سازی الگوریتم MARCO (تحلیل^{۷۶} SBAC) یا الگوریتم ESPE (تحلیل^{۷۷} RBAC) برنامه های به زبان جاوا را تحلیل می کند. [۳۷]

۲-۱-۳ روش فازینگ

تعاریف مختلفی از آزمون فازینگ در مقاله های مختلف بیان شده است. از جمله: «روش آزمون با داده های دلخواه»، «آزمون جعبه سیاه خودکار» یا «روش آزمون خودکار با داده های ورودی مختلف به منظور کشف آسیب پذیری برنامه». به طور کلی می توان این روش را بر اساس مراحل اجرای آن تعریف کرد. ابتدا تولید ورودی دلخواه سپس اجرای برنامه با این ورودی ها و در ادامه بررسی این که آیا برنامه با این ورودی ها متوقف می شود یا نه.

ابتدا برنامه با داده های ورودی کاملاً دلخواه اجرا می شد که پوشش کاملی از تمام مسیرهای برنامه نداشت. بعد از آن محققان دو روش برای تولید ورودی برنامه ها پیشنهاد دادند: (۱) روش تولید-داده (۲) روش جهش-داده. در روش اول، داده های ورودی بر اساس تعریف ورودی برنامه ایجاد می شود. مثلاً اگر ورودی به شکل فایل باشد، ورودی ها بر اساس قالب فایل ورودی ایجاد می شود. 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، راه کار دوم بهتر است. در این مورد با توجه به ورودی مسیر خاصی از برنامه اجرا می شود. ابزارهای Autodafe و APIKE Proxy از روش دوم استفاده می کنند. ابزار Peach ترکیب هر دو روش است. ابزارها از [۳۶] معرفی شده اند.

۳-۱-۳ تحلیل پویا

در مقاله [۳۸] تحلیل پویا به شکل زیر تعریف شده است:

تعریف ۴: تحلیل پویا، تحلیل خاصیت^{۷۸} های مورد نظر یک برنامه ی در حال اجرا را گویند.

⁷⁶ Stack Based Access Control

⁷⁷ Role Based Access Control

⁷⁸ Property

بر خلاف تحلیل ایستا، که با بررسی کد برنامه به دنبال بررسی خاصیت‌ها برای تمام اجراهای برنامه است، در تحلیل پویا با اجرای برنامه به دنبال بررسی خاصیت‌ها در یک یا دو اجرا از برنامه است که معمولاً با تجهیز (یا نظارت‌مند کردن)^{۷۹} برنامه حاصل می‌شود. تحلیل پویا نمی‌تواند برقراری یک خاصیت را اثبات کند ولی در مقابل می‌تواند عدم برقراری آن خاصیت را نشان دهد. از ویژگی‌های این تحلیل می‌توان به دو مورد زیر از مقاله [۳۸] اشاره کرد:

- دقت اطلاعات: تحلیل پویا معمولاً همراه با تجهیز (یا نظارت‌مند کردن) کد برنامه است که با آن معین می‌شود در زمان اجرا چه اطلاعاتی از وضعیت برنامه نگهداری شوند. این اطلاعات می‌توانند در جهت یک تحلیل خاص جمع‌آوری و محدود شوند. برای مثال در تحلیل پویای خودمان، ما اطلاعات مورد نیاز برای تشخیص آسیب‌پذیری SQL را تنها نگهداری می‌کنیم.
- اتکا به ورودی برنامه: تحلیل پویا به ورودی برنامه متکی است. این موضوع موجب می‌شود که این تحلیل کامل نباشد ولی در عین حال مکانیزمی قدرتمند برای ارتباط دادن بین ورودی‌ها و خروجی‌های برنامه است. در این پژوهش با استفاده از اجرای پویا-نمادین سعی کرده‌ایم تا تمام ورودی‌های ممکن و مورد نیاز برای تحلیل را تولید کنیم. در اجرای پویا-نمادین با استفاده از حل‌کننده قید این امر ممکن 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.

۴-۱-۳ 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. آزمون نفوذ در مقاله [۳۶] به سه دسته کلی دسته بندی شده است:

۱. جعبه سیاه^{۸۰}: آزمون نرم‌افزار از بیرون و بدون دسترسی به مستندسازی‌ها و مشخصات آن، آزمون جعبه سیاه گویند.

⁷⁹ Instrument

⁸⁰ Black Box Testing

۲. جعبه سفید^{۸۱}: در بررسی نرم افزارها اگر کد منبع ویا مستندسازیها ویا مدل تهدید در اختیار باشد، به آن آزمون جعبه سفید می گویند.

۳. جعبه خاکستری^{۸۲}: اگر آزمون جعبه سیاه و سفید به صورت ترکیبی استفاده شوند، آزمون جعبه خاکستری خواهیم داشت. در مورد نرم افزارهای مختلف، متخصصان یک بار از دید توسعه دهندگان و از داخل به بیرون کدها و برنامه را می آزمایند. سپس یک بار هم از دید مهاجم و از بیرون به داخل لایه به لایه کدها و برنامه را بررسی می کنند.

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

آسیب پذیری های مختلفی از جمله سرریز بافر، سرریز هیپ، تزریق کد و غیره مطرح هستند. در این پژوهش ما آسیب پذیری تزریق را مورد توجه قرار داده ایم. در بخش بعد این آسیب پذیری بررسی خواهد شد.

۲-۳ آسیب پذیری تزریق

در [۳۹] آمده است که وقتی برنامه کاربردی به نوعی باشد که از طرق یک برنامه واسط، یک دستور به سیستم عامل یا پایگاه داده ارسال شود، امکان وجود این آسیب پذیری وجود خواهد داشت. هرگاه برنامه ای از یک مفسر^{۸۳} از هر نوعی استفاده کند، خطر اینگونه حملات وجود خواهد داشت.

تعدادی زیادی از برنامه های تحت وب از برنامه های خارجی یا فراخوانی های سیستمی برای ارائه کاربردهای خود استفاده می کنند. برنامه Sendmail از پرکاربردترین این برنامه هاست. وقتی اطلاعات از

⁸¹ White Box Testing

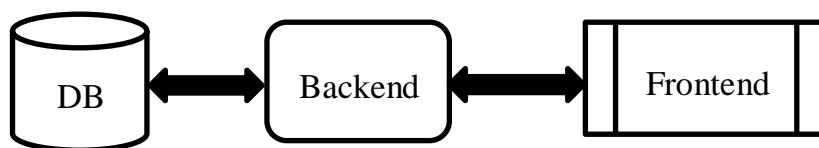
⁸² Gray Box Testing

⁸³ Interpreter

طریق یک درخواست HTTP ارسال می‌شود، داده‌های ارسال شده باید کاملاً 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 پایگاه داده ارسال می‌کند. با این کار و جاسازی کردن یک پرس‌وجوی بدخواه در میان داده‌های ارسالی، مهاجم می‌تواند اطلاعات پایگاه داده 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 مثلاً «`rm -r *`»; برای پاک کردن تمام فایل‌ها به صورت بازگشتی. یا با اضافه کردن دستوراتی در قسمت «where» در پرس‌وجوهای به پایگاه داده می‌توان پرس‌وجوی مورد نظر خود را ارسال و اجرا کرد. مثلاً اضافه کردن «`or 1=1`» به انتهای پرس‌وجو باعث می‌شود پرس‌وجو به ازای هر مقدار «where» درست باشد. همان طور که در شکل ۱-۳ نیز دیده می‌شود، کاربر از طریق قسمت رابط کاربری (قسمت ۱) با برنامه در ارتباط است و داده‌های خود را وارد می‌کند. در قسمت ۲، درخواست کاربر پردازش شده و در صورت نیاز پرس و جویی به پایگاه داده (قسمت ۳) ارسال می‌شود. در صورتی که آسیب‌پذیری تزریق SQL وجود داشته باشد، کاربر می‌تواند ورودی‌های آلاینش شده به برنامه بدهد (قسمت ۱). بعد از پردازش‌های لازم، در صورتی که خروجی پردازش در رابط کاربری نمایش داده شود (قسمت ۱)، کاربر می‌تواند اطلاعات پایگاه داده را بدست بیاورد. نکته مهم این است که حتی اگر خروجی خاصی هم در رابط کاربری نمایش داده نشود، ممکن است آسیب‌پذیری به نحوی باشد که بتوان دستورات خطرناکی مثل «DROP TABLE» و یا «DROP DATABASE» را اجرا و داده‌های موجود در پایگاه داده را به کلی پاک کرد.



شکل ۱-۳ شمای کلی از برنامه‌های دارای پایگاه داده و ارتباط مولفه‌های مختلف با هم.

۱-۲-۳ قسمت‌های آسیب پذیر برنامه

با توجه به راهکاری که در [۳۹] آمده است می‌توان گفت که برای اینکه برنامه‌نویس بتواند قسمت‌های آسیب پذیر در برنامه را تشخیص دهد، باید کد خود را جست‌وجو کند و تمام محل‌هایی را بررسی کند که به صورت مستقیم (مثلاً `system, exec, fork, Runtime.exec` یا پرس‌وجوهای `SQL`) یا غیرمستقیم (تمام درخواست‌های `HTTP`) یک فراخوانی به سیستم یا پایگاه داده ارسال می‌شود. با این بررسی امکان وقوع آسیب پذیری کاهش پیدا می‌کند.

۲-۲-۳ راه‌های مقابله با آسیب پذیری تزریق

در [۳۹] راه‌های مقابله با آسیب پذیری تزریق آمده است. در ادامه این راهکارها آمده است:

- استفاده از توابع کتابخانه‌ای موجود به جای فراخوانی‌های سیستمی.
- بررسی اعتبار 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^{۸۴} را روی سیستم عامل نداشته باشد.
- استفاده از توابع تصفیه‌کننده^{۸۵}. مثل `htmlspecialchars` 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.
- استفاده از `Stored Procedures`ها به جای نوشتن پویای پرس‌وجو در رابطه با پایگاه داده‌ها.

در شکل ۲-۳ 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

⁸⁴ Root

⁸⁵ Sanitizer Methods

نمایش خروجی هم وجود نداشته باشد، باز مهاجم با تزریق «"; DROP TABLE user_data#» می تواند با پاک کردن تمام اطلاعات یک جدول، آسیب برساند.

```
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 ۲-۳ نمونه کد ناامن به زبان جاوا برای آسیب پذیری تزریق کد SQL

این موضوع اهمیت بررسی اعتبار ورودی کاربر را نشان می دهد. در شکل ۳-۳ نمونه امن همان کد آمده است. در آن کد از شکل پارامتری برای گرفتن ورودی از کاربر استفاده می شود. در این حالت ورودی کاربر در محل «?» در پرس و جو قرار می گیرد. معتبر بودن عبارت وارد شده هم توسط کتابخانه مربوط به آن بررسی می شود. در این حالت مهاجم نمی تواند کد دلخواه خود را تزریق کند.

```
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 ۳-۳ نمونه کد امن به زبان جاوا برای آسیب پذیری تزریق کد SQL

۳-۳ آسیب پذیری در برنامه‌های اندرویدی

در این بخش به طور خاص در مورد آسیب‌پذیری‌های موجود در برنامه‌های 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 برنامه‌های اندرویدی و توابع آسیب‌پذیر در آن را همراه با مثال خواهیم دید.

۱-۳-۳ آسیب‌پذیری‌های تزریق در برنامه‌های اندرویدی

آسیب‌پذیری‌های متنوعی در برنامه‌های اندرویدی وجود دارند. از جمله می‌توان آسیب‌پذیری تزریق، نشت اطلاعات حساس کاربر، ربودن مولفه^{۸۶}، نشت توکن توانایی^{۸۷}، پیمایش دایرکتوری‌ها در ContentProvider^{۸۸}، پیاده‌سازی آسیب‌پذیر^{۸۹} SSL، مجوز دسترسی عمومی به فایل^{۹۰} و ثبت کردن اطلاعات حساس^{۹۱} را نام برد. در این پژوهش، از میان آسیب‌پذیری‌های گفته شده، توجه ما به آسیب‌پذیری‌های تزریق است. در این قسمت به طور خلاصه آسیب‌پذیری‌های تزریق در برنامه‌های اندرویدی توضیح داده می‌شوند.

بر اساس کارهای گذشته، ۴ نوع آسیب‌پذیری تزریق در برنامه‌های اندرویدی دیده شده است. این آسیب‌پذیری‌ها عبارتند از: آسیب‌پذیری تزریق SQL (بخش ۳-۳-۳)، تزریق دستور به پوسته سیستم عامل [۱۷]، تزریق کد جاوا اسکریپت به WebView [۴۰] و تزریق Intent به برنامه [۴۱].

در بعضی از برنامه‌های اندرویدی برنامه‌نویس با استفاده از API‌های موجود، دستورات سیستم عاملی را اجرا کرده و از نتیجه آن بهره می‌برد. همان طور که در شکل ۳-۴ دیده می‌شود، برنامه‌نویس با دریافت ورودی از کاربر، دستور ping را با اجرای یک پردازش سیستمی اجرا می‌کند. مشکل زمانی ایجاد می‌شود

⁸⁶ Component Hijacking

⁸⁷ Capability Leak

⁸⁸ ContentProvider Directory Traversal

⁸⁹ X509TrustManager Implemented Improperly

⁹⁰ Public File Access Permission

⁹¹ Log Sensitive Information

که برنامه نویسی ورودی کاربر را بررسی نکند. زیرا می توان به این دستور، دستورات دیگری را چسباند و آنها را نیز اجرا کرد. مثلاً با دستور "data" && rm -rf "www.remotehostname.com" دایرکتوری data نیز پاک می شود.

```
//Obtain a process running an OS Shell
Process proc = Runtime.getRuntime().exec ("sh -i");
//Obtain a stream to send commands to that process
OutputStream programInput = proc.getOutputStream(); //Obtain some untrusted input
String user input = getUntrustedInput( ); //Create a command with untrusted input, and execute the command
programInput.write ("ping -c 4" + userinput + "\n" );
```

شکل ۳-۴ نمونه کد آسیب پذیر به تزریق دستور پوسته سیستم عامل

بعضی از برنامه های اندرویدی از WebView به عنوان یک مرورگر داخلی در برنامه برای نشان دادن صفحات وب استفاده می کنند. با استفاده از دستور loadUrl() می توان کدهای دلخواه جاوا اسکریپت را با همان مجوزهای کدهای موجود در صفحه وب اجرا کرد. دستورات تزریق شده می توانند درخت DOM صفحه و یا کوکی ها را دستکاری کنند. WebView یک پارامتر javascriptenable دارد که به صورت پیش فرض غیرفعال است. اما به سادگی ممکن است که از طرف برنامه نویسی فعال شود.

```
String js = "javascript: var newscript = document.createElement(\"script\");";
js += "newscript.src=\"http://www.attack.com/malicious.js\"";
js += "document.body.appendChild(newscript);";
mWebView.loadUrl(js);
```

شکل ۳-۵ نمونه کد برای آسیب پذیری تزریق جاوا اسکریپت به WebView

همان طور که در شکل ۳-۵ نیز دیده می شود، کد malicious.js در صفحه بارگذاری شده اجرا می شود. مثلاً اگر صفحه بارگذاری شده facebook باشد، این کد می تواند دوستان کاربر را حذف کند! به طور مشخص اگر برنامه توسط خود facebook پیاده سازی شده باشد، چنین حمله ای ممکن نیست. ولی برنامه های غیر رسمی در بازار وجود دارند، که امکان اتصال به صفحه facebook را می دهند.

همان طور که در فصل ۱ توضیح داده شد، Intent مکانیزم انتقال پیام در اندروید است و برنامه های زیادی از آن استفاده می کنند. ممکن است در برنامه ای از مولفه برنامه نویسی دیگر استفاده شود. اگر این

مولفه به درستی مجوزدهی نشود، امکان آسیب پذیری تزریق Intent وجود خواهد داشت. برای مثال اگر برنامه مودبی روی سیستم وجود داشته باشد و در برنامه قربانی هم مولفه مورد نظر به درستی مجوزدهی نشده باشد، برنامه مودی می تواند Intent تولید کرده و به قربانی ارسال کند. نتیجه این حمله می تواند متفاوت باشد. اگر در برنامه قربانی یک Broadcast Receiver وجود داشته باشد، قربانی ممکن است بر اساس Intent ارسالی بر روی داده های آلاینده عملیاتی انجام دهد. ممکن است یک Activity در برنامه قربانی اجرا شود و حمله CSRF (مشابه آنچه در وب وجود دارد) اتفاق بیفتد. همچنین ممکن است در برنامه قربانی یک سرویس اجرا شود و برنامه مودی با استفاده از آن سرویس به اطلاعات حساس کاربر دسترسی پیدا کند.

در این بخش آسیب پذیری های موجود در برنامه های اندرویدی معرفی شدند. به طور خاص آسیب پذیری تزریق به تفصیل مورد بررسی قرار گرفت. در بخش بعد ابزارهایی معرفی خواهند شد که برای تشخیص این آسیب پذیری ها ارائه شده اند.

۲-۳-۳ تشخیص آسیب پذیری در برنامه های اندرویدی

برای تشخیص آسیب پذیری در برنامه های اندرویدی رویکردهای متفاوتی وجود دارد. تا به حال از روش های ایستا بیشتر از روش های پویا استفاده شده است. در میان روش های پویا تا به حال از روش پویا-نمادین برای تشخیص آسیب پذیری استفاده نشده است. در [۱۰] دسته بندی کامل و جامعی از تمام روش های آزمون امنیتی در حوزه اندروید آمده است. این کار ادعای ما مبنی بر نوآوری ما در استفاده از اجرای پویا-نمادین در تشخیص آسیب پذیری را تایید می کند. بر اساس نتایج دسته بندی این کار تنها ابزارهایی که در فصل پیش به تفصیل توضیح داده شدند از روش پویا-نمادین استفاده می کنند و همان طور که گفته شد، تا به حال از این روش در تشخیص آسیب پذیری برنامه ها استفاده نشده است. در ادامه تعدادی از کارهایی آورده می شود که در حوزه اندروید به منظور تشخیص آسیب پذیری ارائه شده اند.

ابزار TaintDroid [۱۱] از تحلیل آلاینده به منظور تشخیص آسیب پذیری نشت اطلاعات حساس کاربر استفاده می شود. ابزار SCanDroid [۱۲] هم با تحلیل ایستا به دنبال نشت اطلاعاتی است که خط مشی امنیتی تعریف شده در تنظیمات برنامه را نقض می کند.

ابزار CHEX [۱۳] در سال ۲۰۱۲ ارائه شده است این ابزار از روشی ایستا برای تشخیص آسیب پذیری «ربودن مولفه» استفاده می کند. این ابزار با استفاده از گراف جریان داده و گراف وابستگی سیستمی این آسیب پذیری را تشخیص می دهد.

در مرجع [۱۷]، 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 آسیب پذیری ها روش دنبال کردن آرایش استفاده شده است. برای پیاده سازی، گفته شده که باید سیستم عامل اندروید تغییر پیدا کند تا بتوان رشته های آرایش یافته را دنبال کرد. ما در این پژوهش بدون تغییر سیستم عامل به این هدف خواهیم رسید.

در [۱۶] ابزاری ارائه شده است که به وسیله آن می توان آسیب پذیری برنامه اندرویدی به افشای اطلاعات حساس از طریق Intent (که در فصل ۱ معرفی شد) را تشخیص داد. در این کار اطلاعات مربوط به Intent ها و مجوزهای آنها با تحلیل ایستای فایل Manifest.xml برنامه استخراج می شود. سپس در یک محیط شبیه ساز برنامه اجرا شده و گردش Intent ها دنبال می شوند.

در ابزار APSET [۱۴] با روش ایستا به دنبال آسیب پذیری های مبتنی بر Intent است. این ابزار با استفاده از الگوی آسیب پذیری ها و دیگرام کلاس های برنامه ویژگی های آسیب پذیری که باید آزمایش شوند را استخراج می کند سپس بر همین اساس برای برنامه مورد نظر مورد آزمون می سازد.

در ابزار VulHunter [۱۵] با استفاده از تحلیل ایستا به دنبال تشخیص آسیب پذیری های نشت توکن توانایی، پیمایش دایرکتوری ها در ContentProvider، پیاده سازی آسیب پذیر SSL، مجوز دسترسی عمومی به فایل و ثبت کردن اطلاعات حساس است. در این کار با استفاده از گراف کنترل جریان بین تابعی، گراف وابستگی سیستمی و گراف فراخوانی توابع گراف توصیف برنامه استخراج می شود. در این کار آسیب پذیری های گفته شده مدل شده اند و با استفاده از این مدل و گراف توصیف برنامه این آسیب پذیری ها تشخیص داده می شوند.

۳-۳-۳ آسیب پذیری تزریق SQL در برنامه های اندرویدی

برنامه های اندرویدی می توانند از پایگاه داده داخلی SQLite استفاده کنند. SDK کتابخانه ای به همین منظور ارائه داده است که امکاناتی برای پیاده سازی امن و همچنین ناامن در آن وجود دارد. در شکل ۳-۶ نمونه برنامه ای با شکل استفاده ناامن (خط ۲) و در شکل ۳-۷ استفاده امن (خط ۳) از این تابعها آورده شده است. در شکل ۳-۶ ابتدا ورودی از کاربر توسط editText گرفته می شود و به تابع query برای پرس و جو داده می شود. سپس خروجی در intent ذخیره شده و به secondActivity فرستاده می شود. در شکل ۳-۷ نام دانشجو توسط editText گرفته می شود و سپس از پایگاه داده پاک می شود. در [۳۷] راه های وقوع و مقابله با این آسیب پذیری به صورت کلی آمده است. از جمله راه کارهای مقابله، استفاده از تابع های پارامتری است که در SDK این امکان وجود دارد. هرگاه برنامه نویس از شکل پارامتری استفاده می کند، به جای مقادیری که قرار است کاربر وارد کند از نویسه «؟» استفاده می کند و ورودی های کاربر را به عنوان ورودی تابع آسیب پذیر قرار می دهد. (خط ۳ در شکل ۳-۷) با این کار ورودی کاربر به طور مستقیم با دستور برنامه نویس جمع نمی شود بلکه در پایگاه داده ابتدا دستور SQL تحلیل و درخت تجزیه^{۹۲} آن تولید می شود. بعد از آن ورودی کاربر به عنوان یک رشته به جای ؟ قرار می گیرد. این کار از تجزیه شدن ورودی کاربر در پایگاه داده جلوگیری می کند. در جدول ۳-۱ مجموعه ای از تابع های آسیب پذیر در SDK آورده شده است. در صورتی که درست از آنها استفاده نشود، می توانند موجب آسیب پذیری تزریق SQL شوند.

⁹² Parse Tree

```

1:public void onClick(View v){
2: Cursor c = db.query(false, "student", null, "stdno=" + editText.getText().toString()
+ "", null, null, null, null, null);
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 ۳-۶ نمونه کد آسیب پذیر در استفاده از SQLite در اندروید

```

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 ۳-۷ نمونه کد امن در استفاده از SQLite در اندروید

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

جدول ۱-۳ تابع‌های آسیب پذیر به تزریق SQL در اندروید

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

۴-۳ جمع بندی

در این فصل به تفصیل مطالعه‌ای بر روی آسیب پذیری‌های نرم‌افزاری انجام دادیم. همچنین آسیب پذیری تزریق را به طور خاص و همراه با مثال بررسی کردیم. در ادامه فصل آسیب پذیری‌های مربوط به برنامه‌های 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 آسیب پذیری در برنامه‌های اندرویدی استفاده نشده است. این روش از آنجایی که می‌تواند به پوشش بالایی از کد برنامه برسد، می‌تواند آسیب پذیری‌های مختلف را بهتر تشخیص دهد. چون که آسیب پذیری‌ها 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 برنامه اجرا می‌شود پس مثبت نادرست

در تشخیص وجود نخواهد داشت و این موضوع تحلیل را دقیق تر می کند. البته اجرای پویا-نمادین در برنامه‌های بزرگ در دنیای واقعی با مسئله انفجار مسیر روبه‌رو می شود پس نیاز است تا این روش در ترکیب با یک روش ایستا ارائه شود تا نیاز نباشد تا تمام مسیرهای موجود در درخت اجرای برنامه به صورت پویا-نمادین اجرا شوند. در فصل بعد راه‌کاری برای تشخیص آسیب‌پذیری در برنامه‌های اندرویدی با استفاده از اجرای پویا-نمادین پیشنهاد خواهد شد.

۴

فصل چهارم راه کار پیشنهادی

راه کار پیشنهادی

در فصل اول چالش‌های مربوط به تحلیل برنامه‌های اندرویدی توضیح داده شد. در این فصل ما برای مقابله با هر کدام از چالش‌های مطرح شده و بهینه شدن فرایند تحلیل، ایده‌ها و نوآوری‌های خود را ارائه خواهیم کرد. در این فصل با ایده کلاس‌های Mock، چالش رخدادمحور بودن برنامه‌های اندرویدی و همچنین درهم‌تنیدگی زیاد با SDK را حل کرده‌ایم. علاوه بر آن به کمک این کلاس‌ها و البته تحلیل ایستا و الگوریتم خودمان، چالش نبود موتور اجرای پویا-نمادین را با به کارگیری و تغییر SPF حل کرده‌ایم. لازم به ذکر است که در این پژوهش سعی کرده‌ایم تا مسئله انفجار مسیر را با اضافه کردن تحلیل ایستا و روش هیوریستیک خودمان بهبود ببخشیم. در کار انجام شده، ما SPF را گسترش داده‌ایم تا با ترکیب اجرای پویا-نمادین و تحلیل آرایش و ارائه ایده کلاس Mock نمادین، بتوانیم آسیب‌پذیری تزریق را در برنامه‌های اندرویدی تشخیص دهیم.

به طور خلاصه در این پژوهش ما بر روی دو موضوع کار کرده‌ایم:

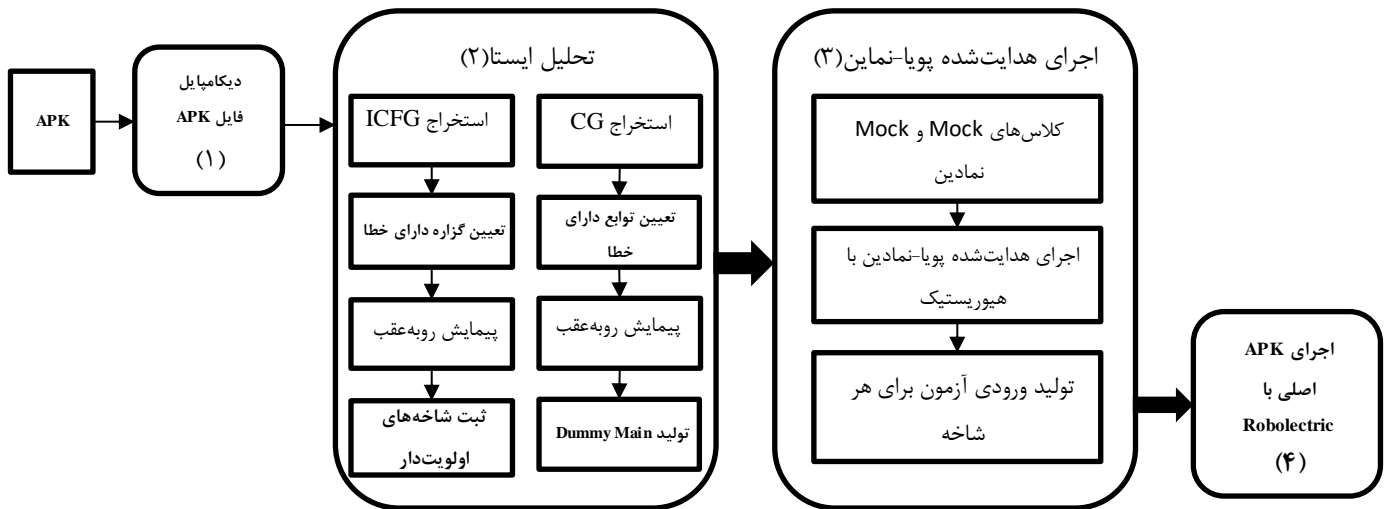
- ارائه یک روش هیوریستیک برای اجرای پویا-نمادین هدایت شده

- اجرای پویا-نمادین برای تشخیص آسیب‌پذیری تزریق در برنامه‌های اندرویدی

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 و بهبود مسئله انفجار مسیر در برنامه‌های 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 برنامه‌های اندرویدی را ارائه خواهیم داد.

لازم به ذکر است که بهبود اجرای پویا-نمادین و ارائه یک روش هیوریستیک برای این منظور جزئی از پیشنهاد پروژه اولیه نبوده است. این موضوع به صورت جداگانه طرح و تعریف شده است.

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

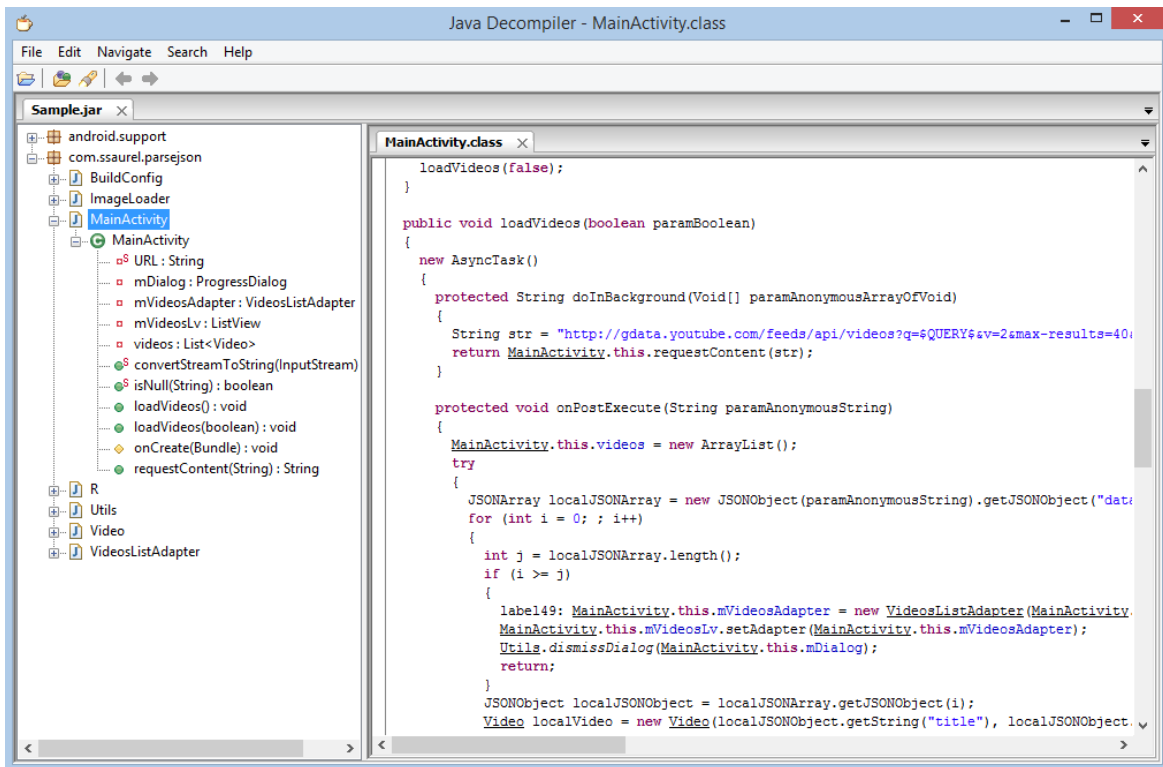


شکل ۴-۱ معماری کلی طرح پیشنهادی

معماری کلی ارائه شده ما برای طرح پیشنهادی در شکل ۴-۱ نشان داده شده است. معماری ابزار ما از چهار بخش کلی تشکیل شده است. در بخش (۱) فایل apk برنامه را دیکامپایل می‌نماییم، و سپس در بخش (۲) به تحلیل ایستا آن می‌پردازیم. خروجی تحلیل ایستا، تعیین نقطه ورودی برنامه و پشته حاوی شاخه‌های اولویت‌دار است. در بخش (۳)، با استفاده از خروجی تحلیل ایستا، اجرای پویا-نمادین همراه با روش هیوریتیک ارائه شده اعمال می‌گردد، تا ورودی آزمون را به دست آوریم. در بخش (۴)، برنامه اصلی را با ورودی‌های عینی و ابزار Robolectric می‌آزماییم. در ادامه بخش‌های مختلف معماری کلی شرح داده می‌شود.

۴-۱-۱ دیکامپایل برنامه

در این بخش، ما با استفاده از ابزار APKTool فایل apk برنامه را دیکامپایل می‌نماییم. خروجی این بخش در واقع Activityهای برنامه است که به صورت فایل java تولید می‌گردد. خروجی دیکامپایل یک برنامه نمونه در شکل ۴-۲ آمده است. فایل APK از حالت فشرده خارج می‌شود. سپس فایل باینری dex به فایل Jar تبدیل می‌شود. سپس با استفاده از ابزار Decompile زبان Java کد برنامه بدست می‌آید.



شکل ۴-۲ خروجی دیکامپایل برای یک برنامه نمونه

۲-۱-۴ تحلیل ایستا

در کار ما تحلیل ایستا به دو منظور انجام می‌شود. از آنجایی که موتور اجرای پویا-نمادین برای برنامه‌های 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 برنامه‌های اندرویدی این گونه نیستند. در بخش تحلیل ایستا ابتدا نقطه ورودی برنامه را با تحلیل «گراف فراخوانی توابع»^{۹۳} 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»^{۹۴}، 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.

^{۹۳} Call Graph (CG)

^{۹۴} Inter- Control Flow Graph (ICFG)

۴-۱-۲-۱ استخراج نقطه ورودی برنامه

برنامه‌های اندروید برخلاف برنامه‌های دیگر چندین نقطه شروع دارند و از تابعی مشخص شروع نمی‌شود [۲]. در واقع یک برنامه اندروید می‌تواند چندین نقطه شروع داشته باشد که با توجه به رخدادهای متفاوت ایجاد شده، برنامه از یکی از آن نقطه‌ها آغاز می‌شود. مثلاً یک برنامه با آمدن یک رخداد مثل دریافت یک پیام ممکن است کار خود را شروع کند یا همان برنامه با باز کردن عادی آن و مثلاً فشردن یک دکمه کار خود را آغاز 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 برنامه 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 روش هیوریستیک و تابع نقطه ورودی برنامه، «استثنای زمان اجرا»^{۹۵} را انتخاب کرده‌ایم. توجه گردد برای اینکه سایر خطاها مانند «خطای نشت حافظه» را نیز بتوانیم کشف کنیم صرفاً کافی است تحلیل ایستای متناسب با آن به ابزار اضافه شود.

```

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 ۴-۳ نمونه تابع نقطه شروع برنامه برای MunchLife

⁹⁵ Runtime Exception

استثنای زمان اجرا می‌تواند از جنس «خطای تقسیم بر صفر»، «استثنای نقض محدوده آرایه» یا موارد دیگری باشد که در زمان اجرای برنامه اعلام^{۹۶} می‌شود. در برنامه‌های مورد آزمون، در نقاط مناسب برنامه، کد تولیدکننده این استثنا را قرار می‌دهیم.

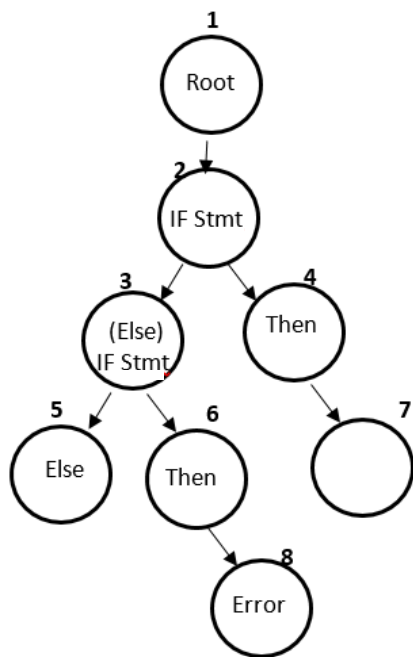
برای تولید تابع نقطه ورودی به برنامه باید گراف فراخوانی توابع را پیمایش نمود. اگر این گراف را به صورت روبه‌جلو و کامل پیمایش کنیم، می‌توانیم به حداکثر پوشش کد دست یابیم. اما با توجه به اینکه یافتن خطا مهمتر از پوشش حداکثری کد است، ما در اینجا ایده پیمایش روبه‌عقب گراف فراخوانی توابع، از تابع دارای خطا به ریشه را مطرح می‌کنیم. گراف فراخوانی توابع را با ابزار Soot [۴۲] به دست آورده‌ایم. سپس الگوریتم پیمایش رو به عقب پیشنهادی خودمان را روی گراف استخراج شده اعمال کرده‌ایم. نمونه تابع نقطه شروع به برنامه در شکل ۳-۴ دیده می‌شود.

۲-۲-۱-۴ تعیین پشته شاخه‌های اولویت‌دار

برای به دست آوردن اولویت اجرای هر شاخه، گراف کنترل جریان بین تابعی برنامه 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 ۴-۴ نمونه این گراف آمده است. در این گراف از گره خطا (گره ۸) به صورت روبه‌عقب پیمایش به سمت گره ریشه (گره ۱) آغاز می‌کنیم. در گرهی که دستور شرطی وجود دارد، دستور شرطی همراه با اینکه شاخه then بر else اولویت دارد را در پشته ذخیره می‌کنیم. در این مثال در گره ۳، دستور شرطی و اولویت شاخه then بر else را به پشته اضافه می‌کنیم. همچنین برای گره ۲، دستور شرطی و اولویت else بر then را به پشته اضافه خواهیم کرد.

⁹⁶ Notification



شکل ۴-۴ مثالی از گراف کنترل جریان بین

لازم به ذکر است که در اینجا منظور از خطا (Error) آسیب پذیری تزریق SQL نیست. در اینجا برای اینکه بهتر شدن انفجار مسیر در اجرای پویا-نمادین را بررسی کنیم، از استثنای زمان اجرا استفاده کرده ایم. هدف از ما از اجرای این تحلیل بهبود چالش انفجار مسیر است.

۳-۱-۴ تولید کلاس‌های Mock و Mock نمادین

نمادین

برای اینکه برنامه روی JVM قابل اجرا باشد و چالش رخدادمحور بودن را حل کنیم، از کلاس‌های Mock به جای کلاس‌های اصلی SDK استفاده کرده ایم. 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 برنامه با فراخوانی تابع Mock مرتبط با آن شبیه‌سازی می‌کنیم.

همان طور که پیش از این گفته شد، 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. ایده کلاس‌های Mock از ایده استفاده از «خلاصه تابع»^{۹۷} به جای تابع اصلی در برنامه گرفته شده است. همان طور که در [۲۰] هم آمده است، برای اینکه اجرای پویا-نمادین با سرعت بیشتر و به صورت بهینه و هدفمند اجرا شود، خلاصه‌ای از تابع‌هایی که هدف آزمون نیستند، (مثل تابع‌های کتابخانه‌ها) تولید می‌شود. این خلاصه، عطف^{۹۸} همه قیدهای موجود در آن تابع است. با این کار با هر فراخوانی از این نوع تابع‌ها، به جای اجرای آنها، خلاصه آن تابع با شرط مسیر برنامه تا آن نقطه عطف می‌شود تا مسیرهای ممکن موجود در آن تابع هم در نظر گرفته شوند. در ایده کلاس‌های Mock خلاصه تابع‌ها، یک مقدار همواره درست در نظر گرفته می‌شود.

⁹⁷ Function Summary

⁹⁸ AND

با این کار اجرای مسیرهای مختلف در تابع نادیده گرفته می شود. چون فرض کرده ایم که این تابعها اشکال امنیتی یا نرم افزاری ندارند. برای اینکه خلاصه این توابع همیشه درست باشد، در پیاده سازی، بدنه این تابعها را حذف کرده ایم.

برای اینکه بتوانیم ورودی های آزمون را تولید کنیم، لازم است تا کلاس های از SDK که از کاربر داده دریافت می کنند، (مثل EditText) را به شکل Mock نمادین تولید کنیم. Mock نمادین کلاس Mock است که تمام متغیرها و تمام خروجی های تابع های آن به شکل نمادین هستند. این کار باعث می شود تا به درستی ورودی های که قرار است کاربر وارد کند، بعد از اجرای پویا-نمادین بدست بیایند. برای مثال کلاس Mock مرتبط با EditText در شکل ۴-۵ آمده است. همان طور که در شکل دیده می شود

```
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 ۴-۵ نمونه کلاس Mock نمادین تولید شده برای دریافت ورودی نمادین.

خطوطی که پررنگ نشان داده شده‌اند، موجب شده‌اند که کلاس EditText به صورت Mock نماین تولید شود. رشته Content در تابع سازنده^{۹۹} به صورت رشته نمادین مقداردهی اولیه می‌شود. همین مقدار نمادین در تابع getText باز می‌گردد. همان طور که دیده می‌شود با ایده Mock نمادین لازم نیست تا تعامل کاربر برای وارد کردن ورودی نمادین وجود داشته باشد و این موضوع چالش رخدادمحور بودن در برنامه‌های اندرویدی را حل می‌کند.

۴-۱-۴ اجرای پویا-نمادین هدایت شده با روش هیوربستیک

از مشکلات جدی که اجرای پویا-نمادین با آن روبه‌رو است مشکل انفجار مسیر می‌باشد. در واقع 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 روش هیوربستیک را معرفی می‌کنیم که از انفجار مسیر در اجرای پویا-نمادین برنامه‌های اندرویدی جلوگیری می‌کند. در این روش هیوربستیک راهکار پیشنهادی ما بر دو ایده استوار است:

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

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

ایده الف را در بخش ۴-۱-۲-۱ استخراج نقطه ورودی برنامه و ایده ب را در بخش ۴-۱-۲-۲ 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 عمق اول 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

^{۹۹} Constructor

از تحلیل ایستا استفاده کرده ایم. ما با تحلیل ایستای پیشنهادی خودمان نقطه شروع به برنامه (بخش استخراج نقطه ورودی برنامه) و پشته شاخه های اولویت دار (بخش تعیین پشته شاخه های اولویت دار) را بدست آورده و آنها را به عنوان ورودی به بخش اجرای پویا-نمادین می دهیم.

برای این که اجرای پویا-نمادین متناسب با اولویت های انتخاب شاخه ها صورت پذیرد، الگوریتمی را ارائه داده ایم که به جای پیمایش عمق اول¹⁰⁰، در هر دستور شرطی نظیر حلقه ها و پرش های شرطی، با استفاده از پشته تصمیم می گیریم که اولویت اجرا را به کدامیک از شاخه های پیش رو بدهیم. استفاده از این ایده باعث می شود 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 می کند که اولویت اجرا با شاخه then یا else است. به صورت پیش فرض این تابع همیشه شاخه else را انتخاب می کند و در نتیجه اجرا به صورت عمق اول خواهد بود. ما این تابع را بازنویسی کرده ایم. در این تابع با استفاده از اطلاعات موجود در پشته شاخه های اولویت دار، اولویت اجرای هر یک از شاخه ها را یافته و اعمال می کنیم. دلیل اینکه در تحلیل ایستا داده ها را در پشته ذخیره کرده بوده ایم این است که در تحلیل ایستا گراف مربوطه را به صورت روبه عقب پیمایش می کنیم ولی در SPF و اجرای پویا-نمادین درخت اجرا به صورت روبه جلو پیمایش می شود. پس اطلاعات دستور شرطی در سر پشته، اطلاعات مربوط به اولین دستور شرطی است که در اجرای پویا-نمادین با آن روبه رو خواهیم شد. با استفاده از اجرای پویا-نمادین در نهایت ورودی های آزمون مرتبط با خطاهای موجود در برنامه استخراج خواهند شد.

لازم به ذکر است، روشی که ما ارائه کرده ایم در تشخیص استثنای زمان اجرا در صورتی که تعدادی کمی از آنها در برنامه حضور داشته باشند، موثر است. با فرض وجود این خطا در تمام مسیرهای ممکن و موجود در برنامه، عملاً روش هیوریستیک ما بهبودی در اجرای پویا-نمادین ایجاد نخواهد کرد و در واقع باید تمام مسیرهای موجود در درخت اجرا را پیمایش کند. اما نکته ای که وجود دارد این است که در برنامه های واقعی معمولاً تعداد این خطاها محدود بوده و ما بهبود را در اجرای پویا-نمادین خواهیم دید.

¹⁰⁰ Depth First Search

۴-۱-۵ اجرای برنامه با ورودی‌های عینی

پس از اجرای پویا-نمادین هدایت‌شده برای اطمینان از درستی روش و کشف خطا، برنامه را با ورودی‌های عینی بدست آمده از آن اجرا می‌کنیم. در این بخش از کد منبع برنامه استفاده می‌کنیم تا خطا را با اجرای واقعی برنامه نیز کشف و مشاهده کنیم. برای این منظور از ابزار Robolectric [۲۱] استفاده کرده‌ایم. این ابزار یک ابزار آزمون واحدهای^{۱۰۱} برنامه‌های اندرویدی است. با این ابزار می‌توان قسمتی از برنامه 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-نمادین در SPF لازم بود تا نقطه شروع به برنامه تولید شود. با استفاده از اطلاعات استخراج شده در آن مرحله، ورودی مناسب به ابزار Robolectric را تولید می‌کنیم. نمونه‌ای از آن در شکل ۴-۶ آمده است. همان طور که دیده می‌شود خط ۳ از شکل ۴-۳ مشابه خط ۲ از شکل ۴-۶ است. خطوط ۴ و ۵ از شکل ۴-۳ به صورت خودکار در ابزار اجرا می‌شوند.

```

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

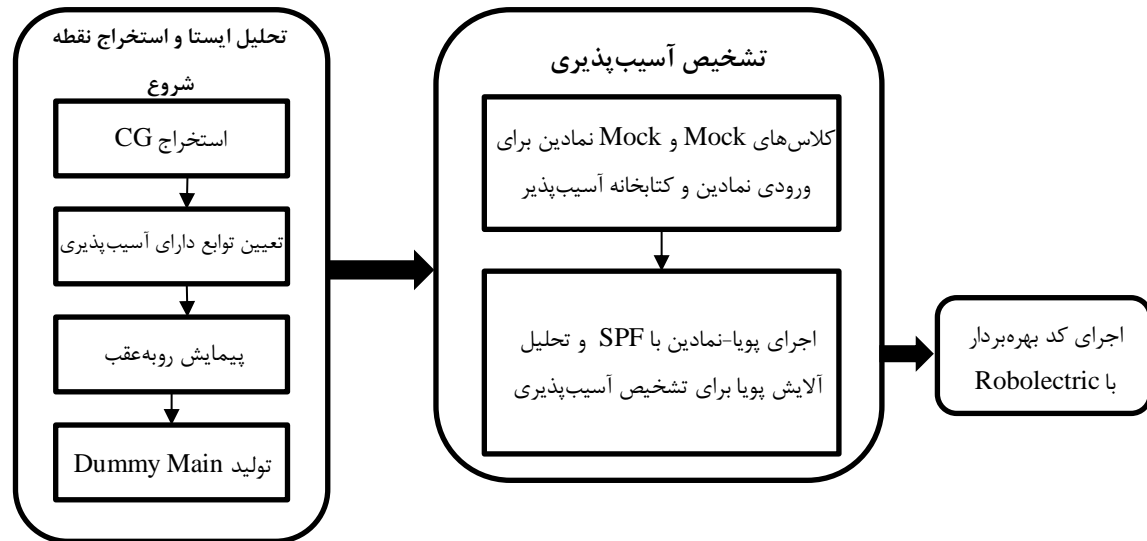
```

شکل ۴-۶ نمونه کد آزمون در Robolectric برای برنامه MunchLife

۴-۲ اجرای پویا-نمادین برای تشخیص آسیب‌پذیری تزریق

در شکل ۴-۷ معماری پیشنهادی ما برای تشخیص آسیب‌پذیری تزریق در برنامه‌های اندروید آورده شده است. برای انجام این کار قسمت‌هایی از معماری که در قسمت قبل توضیح دادیم را در اینجا استفاده کرده‌ایم. در ادامه این معماری به طور کامل توضیح داده خواهد شد.

¹⁰¹ Unit Testing Tool



شکل ۴-۷ فرایند کاری مطابق با معماری کلی طرح پیشنهادی برای تشخیص آسیب پذیری تزریق در برنامه های اندرویدی.

برای تشخیص آسیب پذیری تزریق از تحلیل آرایش همراه با اجرای پویا-نمادین استفاده کرده ایم. در تحلیل ما، هر جا که داده ای نمادین باشد، نشان دهنده این موضوع است که آن داده آرایش شده است. این موضوع باعث می شود تحلیل آرایش با اجرای پویا-نمادین ترکیب شوند. برای اینکه تحلیل آرایش صورت پذیرد لازم است تا ورودی های به برنامه نمادین شوند. برای این منظور ما ایده استفاده از کلاس Mock نمادین را پیشنهاد می کنیم. دلیل استفاده از ایده Mock نمادین این است که تابع های ورودی بخشی از SDK هستند و برنامه نویس صرفاً از آنها استفاده می کند. برای اینکه برنامه اجرا شود لازم است که این کلاس ها Mock شوند و برای اینکه تحلیل کامل شود باید مقادیر متغیرها در آن و خروجی تابع های آن نمادین باشند. سپس برنامه را به شکل پویا-نمادین اجرا می کنیم. در حین اجرا، وضعیت نمادین بودن متغیرها را ذخیره می کنیم. 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 می گیریم. هرگاه اجرا به تابع آسیب پذیر برسد، با توجه به اطلاعات ذخیره شده در رابطه با متغیرها، نمادین بودن ورودی تابع آسیب پذیر را بررسی می کنیم. همچنین در حین اجرا تصفیه شدن ورودی توسط برنامه نویس را بررسی می کنیم. مثلاً اگر $str = str2$ شود str عینی می شود و ما آن را نمادین در نظر نخواهیم گرفت. علاوه بر آن استفاده شدن از تابع پارامتری توسط برنامه نویس به عنوان روشی برای تصفیه کردن داده ها بررسی می شود.

برای اینکه زنجیره آسیب پذیری تزریق کامل شود، لازم است تا خروجی تابع آسیب پذیر به تابع نشت وارد شود. ادامه تحلیل آرایش تا تابع نشت برای برخی از آسیب پذیری های تزریق فرایندی سخت گیرانه

است ولی برای اینکه تحلیل کامل شود و ما بتوانیم تمام آسیب پذیری های تزریق را پوشش دهیم، ما آن را انجام داده ایم. برای اینکه تحلیل آرایش را بتوان ادامه داد، Mock نمادین را برای کلاس آسیب پذیر نیز بایستی تولید کرد. کلاس های آسیب پذیر هم کلاس هایی از SDK هستند و باید به شکل Mock نمادین آنها را تولید کنیم. اجرای پویا-نمادین ادامه پیدا می کند تا زمانی که به تابع نشت برسیم، اگر منشا ورودی به این تابع از کلاس Mock نمادین آسیب پذیر باشد، پس از تابع منبع به تابع آسیب پذیر و سپس به تابع نشت مسیر وجود داشته است. وجود این مسیر یعنی اینکه آسیب پذیری تزریق در برنامه مورد تحلیل وجود دارد.

بر اساس آخرین دانسته ما تاکنون، ابزاری برای تشخیص آسیب پذیری در برنامه های اندرویدی با اجرای پویا-نمادین ارائه نشده است. در منبع [۳۹] عنوان شده است که آسیب پذیری تزریق می تواند موجب نشت اطلاعات حساس کاربر شود. آسیب پذیری تزریق در اندروید می تواند تزریق دستور به پوسته سیستم عامل، تزریق دستورات SQL به پایگاه داده SQLite، تزریق کد جاوا اسکریپت به WebView و یا تزریق Intent باشد. در این پژوهش روش و ابزاری ارائه کرده ایم که می تواند انواع آسیب پذیری تزریق را تشخیص دهد. اگر چه، برای نمونه و نشان دادن درستی کار ابزار، ما آسیب پذیری تزریق SQL را مورد توجه قرار داده ایم.

برای تشخیص سایر آسیب پذیری های تزریق لازم است، تابع های آسیب پذیر مرتبط با آن مطالعه و شناسایی شود. همچنین کلاس Mock نمادین مرتبط با آن تولید شود. تحلیل ایستا نیز باید با توجه به تابع های آسیب پذیر مطالعه شده تغییر یابند. در اجرای پویا-نمادین نیز باید قاعده خاص آن آسیب پذیری تزریق را در آن پیاده سازی کرد. همان طور که دیده می شود، روند کلی تشخیص برای تمام آسیب پذیری های تزریق یکسان است.

۱-۲-۴ تحلیل ایستا

همان طور که گفته شد، برنامه های اندرویدی همانند برنامه های 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 کرده ایم. برای بدست آوردن این گراف از ابزار soot استفاده می کنیم. 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 تا جایی 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 شیء 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 ۴-۸)

```

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 ۴-۸ 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 برنامه را Mock می‌کنیم. در هر دوی این حالت‌ها وجود کلاس Mock هزینه و سربار تحلیل را کاهش می‌دهد.

همان طور که گفته شد، برای اینکه تحلیل آرایش ما به شکل کامل صورت پذیرد، باید کلاس مربوط به تابع آسیب پذیر و تابع منبع را به شکل Mock نمادین تولید کنیم. در این پژوهش، ما برای اولین بار ایده کلاس Mock نمادین را مطرح می کنیم بدین ترتیب که کلاس Mock نمادین کلاسی است که تابع های کلاس اصلی را دارد با این تفاوت که بدنه تابع حذف می شود و خروجی تابع ها نمادین خواهند بود. (در بخش ۴-۱-۳ به طور کامل این ایده مطرح شده است.) این ایده باعث می شود تحلیل آرایش و گردش داده های آرایش شده در برنامه صورت بگیرد و وابستگی به چارچوب کاری اندروید و یا برنامه های دیگر حذف شود. چون کلاس های منبع و آسیب پذیر هر دو بخشی از SDK هستند، پس از این جهت به شکل Mock پیاده سازی می شوند. همچنین چون مقادیر متغیرها و خروجی تابع های آنها در تحلیل آرایش استفاده می شوند، (یعنی داده هایی آرایش شده هستند) پس باید این مقادیر و خروجی ها را نمادین کنیم. نمونه کلاس Mock نماین برای تابع منبع در شکل ۴-۵ آمده است. بخشی از کلاس Mock نماین برای تابع آسیب پذیر کلاس SQLiteDatabase در شکل ۴-۹ دیده می شود. همان طور که دیده می شود تمام خروجی های تابع ها نمادین تولید می شوند.

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

شکل ۴-۹ تکه کدی از کلاس Mock نمادین تولید شده برای کلاس SQLiteDatabase

۴-۲-۳ اجرای پویا-نمادین همراه با تحلیل آرایش

برای تشخیص آسیب پذیری ورودی های برنامه (مثلا EditText) را نمادین در نظر گرفتیم. همچنین یک مولفه جدید به SPF اضافه کردیم. در این مولفه اجرای برنامه را ادامه می دهیم تا زمانی که به تابع آسیب پذیر (مثلا query) برسیم. قاعده بدست آورده ما برای تشخیص آسیب پذیری تزریق SQL به این شرح است که ابتدا نمادین بودن ورودی تابع آسیب پذیری را بررسی می کنیم. در صورت نمادین بودن، پارامتری نبودن تابع آسیب پذیر را تشخیص می دهیم. اگر ورودی تابع آسیب پذیر نمادین باشد، به این معناست که برنامه نویس متغیر ورودی به تابع را تصفیه نکرده است. در صورتی که در پرس و جوی ایجاد شده نویسه «؟» وجود داشته باشد، به این معناست که برنامه نویس از تابع پارامتری استفاده کرده است و امکان آسیب پذیری وجود نخواهد داشت. در صورت نمادین بودن ورودی تابع و همچنین وجود نداشتن نویسه «؟» در پرس و جو، خروجی تابع آسیب پذیر را نمادین می کنیم (Mock نمادین). سپس اجرا را ادامه می دهیم تا زمانی که به یکی از توابع نشت اطلاعات (مثلا TextView) برسیم. در صورتی که ورودی تابع نشت، از Mock نمادین آسیب پذیر باشد، اطلاعات مورد نیاز برای تحلیل را اعلام می کنیم. این اطلاعات شامل شناسه تابع منبع، شناسه تابع نشت، دنباله پشته برنامه تا تابع آسیب پذیر و تصفیه شدن یا نشدن داده ورودی توسط برنامه نویس است. در شکل ۴-۱۰ نمونه خروجی تولید شده برای یک

```

-----Vulnerability Detection Result-----
---STACK TRACE OF CURRENT APPLICATION RUN FOR CATCHING VULNERABILITY-----
1) android.database.sqlite.SQLiteDatabase.rawQueryWithFactory
   (SQLiteDatabase$CursorFactory,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 ۴-۱۰ نمونه خروجی ابزار برای تشخیص آسیب پذیری تزریق SQL

برنامک آسیب پذیر آمده است.

۴-۲-۴ آزمون نرم افزار برای بررسی میزان بهره جویی

با اطلاعاتی که از تحلیل ایستا و پویا-نمادین بدست آوردیم، برنامه را با ابزار Robolectric می آزماییم. ابزار Robolectric به منظور آزمون برنامک های اندرویدی ارائه شده است که نیاز به اجرای برنامک در محیط اندروید ندارد. برای استفاده از این ابزار باید مسیر مورد آزمون به آن داده شود. در شکل ۴-۱۱ نمونه کد بهره بردار در Robolectric آمده است. برای این مورد ما از خروجی تحلیل ایستا استفاده کردیم و عملاً کلاس dummyMain بدست آمده را به Robolectric دادیم (خط ۳ شکل ۴-۸ با خط ۲ شکل ۴-۱۱ نمونه کد بهره بردار در Robolectric، خط ۵ شکل ۴-۸ با خط ۳ شکل ۴-۱۱ نمونه کد بهره بردار در Robolectric و خط ۶ شکل ۴-۸ با خط ۷ شکل ۴-۱۱ نمونه کد بهره بردار در Robolectric یکی است). همچنین لازم است تا رشته هایی مثل «'a' or '1'='1'» را به ورودی آسیب پذیر برنامه بدهیم که معمولاً موجب بهره جویی از آن می شود. (خط ۶ شکل ۴-۱۱ نمونه کد بهره بردار در Robolectric) برای بررسی بهره جویی، خروجی تابع نشت یعنی TextView را مشاهده

```

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 ۴-۱۱ نمونه کد بهره بردار در Robolectric

کردیم (خط ۸ شکل ۴-۱۱ نمونه کد بهره بردار در Robolectric). برای یافتن شناسه تابع ورودی آسیب پذیر و شناسه تابع نشت از خروجی تحلیل پویا-نمادین استفاده کردیم.

۳-۴ جمع بندی

در این فصل راه کار پیشنهادیمان را در رابطه با تشخیص آسیب پذیری در برنامه های اندرویدی با اجرای پویا-نمادین شرح دادیم. همچنین، در این کار دو سوال پژوهشی مطرح کردیم. در سوال اول به دنبال بهبود اجرای پویا-نمادین در آزمون برنامه های اندرویدی بودیم. برای حل این موضوع از تحلیل ایستای برنامه و گراف فراخوانی توابع و گراف کنترل جریان بین تابعی و ایده پشته شاخه های اولویت دار به منظور هدفمند کردن اجرای پویا-نمادین استفاده کردیم. در سوال پژوهشی دوم به دنبال اجرای پویا-نمادین برای تشخیص آسیب پذیری تزریق در برنامه های اندرویدی بودیم. برای این موضوع از ترکیب تحلیل آرایش با اجرای پویا-نمادین استفاده کردیم. همچنین برای بهبود هزینه و سربار، تحلیل را محدود به توابع مطلوبی از برنامه کردیم که از تحلیل ایستا و گراف فراخوانی توابع بدست آورده بودیم. همچنین ایده Mock نمادین را مطرح کردیم که باعث گردش داده های آرایش شده در برنامه می شود. با ترکیب این موارد زنجیره تشخیص آسیب پذیری تزریق یعنی دنبال کردن داده ها از تابع منبع به تابع آسیب پذیر و از تابع آسیب پذیر تا تابع نشت را کامل کردیم. به عنوان خروجی شناسه تابع منبع، شناسه تابع نشت، دنباله پشته برنامه تا تابع آسیب پذیر و تصفیه شدن یا نشدن داده ورودی توسط برنامه نویس را ارائه کردیم. در فصل بعد راه کار پیشنهادی را مورد آزمون قرار می دهیم و نتایج آزمایش ها و تحلیل آنها را ارائه می کنیم. 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 برنامه‌های اندرویدی و تشخیص آسیب‌پذیری در این برنامه‌ها صحبت کردیم. در مورد آزمون برنامه‌های اندرویدی یک روش 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»^{۱۰۲} 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 برنامه‌های اندرویدی بودیم. برای جواب به این سوال تحلیل آلاینش را به اجرای پویا-نمادین اضافه کردیم. عملاً با در نظر گرفتن مقادیر ورودی به صورت نمادین (همان آلاینش شده در تحلیل آلاینش) و اجرای پویا-نمادین برنامه 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 روش هیوریستیک ارائه شده برای اجرای هدایت شده پویا-

نمادین

ارزیابی راه‌کار ارائه شده در این بخش را در دو مرحله انجام دادیم. ابتدا برای نشان دادن این که روش درست کار می‌کند، ۱۰ برنامه را مطرح و پیاده‌سازی کردیم. این برنامه‌ها را به منظور راستی‌آزمایی

¹⁰² ICFG

تولید تابع نقطه شروع، درستی پشته شاخه‌های اولویت‌دار و همچنین درست بودن فرایند تولید کلاس‌های Mock و درست بودن اجرای پویا-نمادین و ارتباط این مولفه‌ها با هم پیاده‌سازی کردیم. در این برنامه‌ها برای نشان دادن وجود خطا، استثنای زمان اجرا را در آنها قرار دادیم. در این ۱۰ برنامه مرحله به مرحله و از ساده‌ترین حالت تا شکل‌های پیچیده را پیاده‌سازی کردیم. همچنین حالت‌های مختلف جریان داده در برنامه (مثلا استفاده از Intent) را پیاده‌سازی کردیم. با استفاده از این برنامه‌ها مولفه جدید پیاده‌سازی شده در SPF را آزمودیم. این مولفه را برای اضافه کردن روش 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 برنامه‌های واقعی اندروید را دارد؟

۲- ابزار پیشنهادی ما نسبت به Sig-Droid که آخرین ابزار آزمون نظام‌مند برنامه‌های اندرویدی است، چه مزیت‌هایی دارد؟

برای پاسخ به سوالات مطرح شده، چهار برنامه دنیای واقعی را آزمودیم که در جدول ۵-۱ مشخصات آن را آورده‌ایم. این برنامه‌ها همان برنامه‌های مورد آزمون ابزار Sig-Droid نیز هستند. این برنامه‌ها از مخزن F-Droid [۲۲] انتخاب شده‌اند. در جدول ۵-۱ اطلاعات این برنامه‌ها آمده است که میزان پیچیدگی آنها را بر اساس تعداد خط کد و تعداد Activityها نشان می‌دهد. هر چه تعداد خط کد و Activityها بیشتر باشد، برنامه پیچیده‌تر خواهد بود.

ابزار Sig-Droid به منظور ارائه اجرای نمادین در برنامه‌های 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 برنامه 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 [۴۳] مقایسه شده است و نشان داده شده است که این ابزار می‌تواند به پوشش بهتری از کد دست یابد.

در این قسمت از کار، ما به دنبال این موضوع بوده‌ایم که با استفاده از تحلیل ایستا و به کارگیری 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 برنامه‌ک را تشخیص دهیم.

جدول ۱-۵ مشخصات برنامه‌های دنیای واقعی مورد آزمون

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

در جدول ۲-۵ اطلاعات مربوط به تحلیل برنامه‌های جدول ۱-۵ با ابزار Sig-Droid که خودمان مطابق با توصیف مقاله پیاده‌سازی کردیم و کار خودمان را آورده‌ایم. این جدول دو قسمت دارد. در قسمت سمت راست، شباهت Sig-Droid و کار ما هنگامی که یافتن خطا مورد نظر نباشد نشان داده شده است. همان‌طور که در بخش ۴-۱ هم گفته شد، در این قسمت از کار برای نشان دادن اینکه تحلیل ایستا موجب بهبود چالش انفجار مسیر در اجرای پویا-نمادین می‌شود، از استثنای زمان اجرا به عنوان خطا استفاده کرده‌ایم. اگر پیدا کردن خطا و روش هیوریستیک را در کار خودمان غیرفعال کنیم، ابزار ما همانند Sig-Droid خواهد بود. همان‌طور که دیده می‌شود ابزار ما هنگام یافتن خطا و فعال بودن هیوریستیک، در زمان کمتر و با پوشش کمتری از خطوط 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 علاوه بر 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 ۲-۵ نتایج بدست آمده با ابزار ما و Sig-Droid پیاده‌سازی شده توسط ما.

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

در جدول ۲-۵، مقایسه ابزارهای مختلف با کار ما بر اساس معیارهای مطرح شده در مقالات [۳] [۴] [۵] آمده است. در معیارهای مقایسه انتخاب شده به ترتیب روش جست‌وجو، انواع رخدادهای برنامه‌های اندرویدی پشتیبانی شده در ابزار، ترکیب تحلیل ایستا و پویا به عنوان روش هیوریستیک در بهینه‌سازی اجرا و چالش انفجار مسیر مطرح شده‌اند. همان‌طور که پیش از این گفته شد برای آزمون برنامه‌های اندرویدی ۳ روش متداول مطرح است. این روش‌ها عبارتند از: آزمون بر اساس ورودی‌های دلخواه، 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^{۱۰۳} (مانند فشردن یک دکمه) قرار می‌گیرند. لازم به ذکر است که مسئله انفجار مسیر در ابزارهای Monkey و Swifthand مطرح نمی‌شود چون این دو ابزار مسیرهای مختلف موجود در کد را بررسی و اجرا نمی‌کنند. این مورد در جدول با «-» نشان داده شده است. علامت «x» در این ستون به این معنی است که انفجار مسیر با این ابزار اتفاق می‌افتد و علامت «✓» در این ستون به این معنی است که انفجار مسیر اتفاق نمی‌افتد. در ستون ترکیب تحلیل ایستا و پویا علامت «✓» به این معنی است که در ابزار مورد نظر از این تکنیک استفاده شده است و علامت «x» به معنی نبود این تکنیک در ابزار است.

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

همان‌طور که در جدول هم دیده می‌شود، کار ما از روش پویا-نمادین به عنوان یک روش نظام‌مند بهره می‌برد و همان‌طور که پیش از این گفته شد، این روش به نسبت تولید ورودی دلخواه و همچنین اجرای نمادین می‌تواند به پوشش بهتری از کد برسد. در کار ما با استفاده از ایده Mock و Mock نمادین

¹⁰³ GUI

می‌توانیم انواع رخدادهای مطرح را پشتیبانی کنیم. با استفاده از روش هیوریستیک ارائه شده که ترکیب تحلیل ایستا و پویا هست توانستیم با سرعت بیشتر و پوشش کد کمتر خطاها را پیدا کنیم، همین موضوع موجب می‌شود که با چالش انفجار مسیر روبه‌رو نشویم.

۵-۲ ارزیابی تشخیص آسیب‌پذیری تزریق در برنامه‌های اندرویدی

ارزیابی راه‌کار ارائه شده در این بخش را در دو مرحله انجام دادیم. ابتدا برای نشان دادن این که روش درست کار می‌کند ۱۰ برنامه را خودمان پیاده‌سازی کردیم. ۶ برنامه را به منظور راستی‌آزمایی تولید کلاس dummyMain و تابع نقطه شروع، همچنین درست بودن فرایند تولید کلاس‌های Mock و درست بودن اجرای پویا-نمادین و ارتباط این مولفه‌ها با هم پیاده‌سازی کردیم. در این برنامه‌ها برای نشان دادن وجود خطا، استثنای زمان‌اجرا را در برنامه‌ها قرار دادیم. در این ۶ برنامه مرحله به مرحله و از ساده‌ترین حالت تا شکل‌های پیچیده را پیاده‌سازی کردیم. در ۴ برنامه باقی‌مانده آسیب‌پذیری تزریق SQL را به جای استثنای زمان‌اجرا قرار دادیم. در این برنامه‌ها 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 برنامه (مثلا استفاده از Intent و یا استفاده از پایگاه‌داده برنامه دیگر) را پیاده‌سازی کردیم. با استفاده از این برنامه‌ها 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 برنامه‌های واقعی هم درست کار می‌کند، از مخزن F-Droid نیز استفاده کردیم. این مخزن شامل برنامه‌های اندرویدی متن‌باز در موضوعات مختلف است. ۱۴۰ برنامه مختلف را به دلخواه انتخاب کردیم. برای تحلیل ابتدا سعی کردیم برای برنامه‌ها تابع نقطه شروع بسازیم. از این برنامه‌ها، برای ۷ برنامه، 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 برنامه 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 برنامه بدست آمده را به صورت دستی هم تحلیل کردیم که نتایج بدست آمده با نتایج خروجی ابزار مطابقت داشت.

در جدول ۴-۵ مقایسه ابزار ما با ابزارهای مشابه فعلی از ۵ جنبه مطرح در مقالات آمده است [۸] [۹] [۴]. ابزارهای Condroid و AppIntent که در این جدول با کار ما مقایسه شده‌اند همانند کار ما، مربوط به حوزه امنیت هستند. Condroid با استفاده از اجرای پویا-نمادین به دنبال کشف بمب منطقی در برنامه‌های اندرویدی است. AppIntent با اجرای نمادین به دنبال کشف نقض حریم خصوصی توسط برنامه‌های اندرویدی است. از این جهت با کار ما که به دنبال تشخیص آسیب‌پذیری تزریق در برنامه‌ها است، مقایسه شده‌اند. ابزار Sig-Droid هم همان‌طور که گفته شد، به منظور آزمون برنامه‌ها 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. همان‌طور که دیده می‌شود کارهای مورد مقایسه همگی مسئله رخدادمحور بودن برنامه‌های اندرویدی را به گونه‌ای حل کرده‌اند. ابزار AppIntent با استفاده از تحلیل ایستای آرایش 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.

در این جدول علامت «✓» به این معنی است که ویژگی مورد مقایسه در ابزار مورد نظر وجود دارد و علامت «✗» به این معنی است که آن ویژگی وجود ندارد. برای مثال در ابزار AppIntent و کار ما انفجار مسیر رخ نمی‌دهد و از تکنیک ترکیب تحلیل ایستا و پویا استفاده شده است ولی در ابزار Condroid و Sig-Droid انفجار مسیر اتفاق می‌افتد و تکنیک ترکیب تحلیل پویا و ایستا وجود ندارد. AppIntent نقض حریم خصوصی، Condroid بمب منطقی و کار ما آسیب‌پذیری تزریق SQL را تشخیص می‌دهند.

به طور خلاصه، 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 برنامه‌های اندرویدی تشخیص دهد.

جدول ۴-۵ مقایسه ابزار ارائه شده با ابزارهای مشابه موجود

معیار مقایسه ابزار	رخداد محور بودن	عدم انفجار مسیر	ترکیب تحلیل ایستا و پویا	تشخیص بوم منطقی	تشخیص آسیب پذیری تزریق SQL	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 برنامه‌های 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 برنامه‌های 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 برنامه‌های 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 برنامه‌های اندرویدی چالش‌های مرتبط به خود را دارد. از جمله وجود SDK و درهم‌تنیدگی زیاد برنامه با آن، وجود نداشتن نقطه شروع به برنامه، کامپایل شدن روی JVM برخلاف DVM و هم چنین رخدادمحور بودن برنامه‌های 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 همچنین، راه‌کاری برای تشخیص آسیب‌پذیری تزریق به برنامه‌های اندرویدی را پیشنهاد دادیم. هر کدام از این سوالات را مورد ارزیابی قرار دادیم.

نتیجه ارزیابی برای راه‌کار ارائه شده به سوال اول این است که با سرعت بیشتر نسبت به Sig-Droid که آخرین کار در این مورد است، خطا را تشخیص می‌دهیم. برای آنکه بتوانیم سایر خطاهای مرتبط با برنامه‌های 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 برنامه‌ک 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 به کلاس‌های Mock بسیار 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 برنامه‌ها 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 برنامه‌ها به زبان جاوا است و با ابزار پیاده‌سازی شده کنونی می‌توانیم تعداد زیادی از برنامه‌ها 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.

منابع

- [۱] “cafebazar.” [Online]. Available: <http://developers.cafebazaar.ir/fa/>. [Accessed: 19-Dec-2017].
- [۲] 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.
- [۳] “Android Monkey.” [Online]. Available: <https://developer.android.com/guide/developing/tools/monkey.html>. [Accessed: 10-Oct-2017].
- [۴] 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.
- [۵] 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.
- [۶] 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.
- [۷] 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.
- [۸] 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.
- [۹] 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.
- [۱۰] 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.

-
- [١١] 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.
- [١٢] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android," 2009, <http://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf>.
- [١٣] 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.
- [١٤] 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.
- [١٥] 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.
- [١٦] 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.
- [١٧] G. J. Smith, "Analysis and Prevention of Code-Injection Attacks on Android OS," University of South Florida, 2014.
- [١٨] "OWASP Mobile Security Testing Guide - OWASP." [Online]. Available: https://www.owasp.org/index.php/OWASP_Mobile_Security_Testing_Guide. [Accessed: 09-Nov-2017].
- [١٩] 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.
- [٢٠] P. Godefroid, "Compositional Dynamic Test Generation," *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 47--54, 2007.
- [٢١] "Robolectric." [Online]. Available: <http://robolectric.org/>. [Accessed: 01-Jan-2017].
- [٢٢] "F-Droid." [Online]. Available: <https://f-droid.org/>. [Accessed: 10-Oct-2017].
- [٢٣] 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.
- [٢٤] 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.
- [٢٥] 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.
- [٢٦] R. Majumdar and K. Sen, “Hybrid concolic testing,” *Proceedings - International Conference on Software Engineering*, pp. 416–425, 2007.
- [٢٧] 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.
- [٢٨] 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.
- [٢٩] 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.
- [٣٠] 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.
- [٣١] 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.
- [٣٢] 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.
- [٣٣] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, “Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART,” *26th USENIX Security Symposium (USENIX Security 17)*, 2017.
- [٣٤] “Valgrind,” 2018. [Online]. Available: <http://valgrind.org/>.
- [٣٥] L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, and P. Liu, “System Service Call-oriented Symbolic Execution of Android Framework with Applications to Vulnerability Discovery and Exploit Generation,” *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017, pp. 225–238.

-
- [٣٤] 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.
- [٣٧] M. Pistoia, “A Unified Mathematical Model for Stack and Role Based Authorization Systems,” Polytechnic University Brooklyn, NY, USA, 2005.
- [٣٨] T. Ball, “The concept of dynamic analysis,” in *ESEC/FSE-7 Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, 1999, vol. 24, no. 6, pp. 216--234.
- [٣٩] “OWASP-Injection Flaw.” [Online]. Available: https://www.owasp.org/index.php/Injection_Flaws. [Accessed: 01-Nov-2018].
- [٤٠] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, “Attacks on WebView in the Android System,” in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 343–352.
- [٤١] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, “Analyzing inter-application communication in Android,” in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, 2011, pp. 239--252.
- [٤٢] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Oceau, and P. Mcdaniel, “FlowDroid: Precise Context , Flow , Field , Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps,” *PLDI '14 Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 259–269, 2014.
- [٤٣] 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.
- [٤٤] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, “jFuzz: A concolic whitebox fuzzer for Java,” in *NASA Formal Methods Symposium*, 2009.
- [٤٥] 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.

15.eu.thedarken.rootvalidator_2.1.1.apk
 16.uk.org.platitudes.wipefiles-v0.3-2016022214.apk
 17.ai.agusibrahim.xhrlog_1.0.apk
 18.info.zamojski.soft.towercollector_1.14.0.apk
 19.codes.simen.l50notifications_1.12.1.apk
 20.io.github.hidroh.tldroid_10.apk
 21.screen.dimmed.pixelfilter_2017-02-19.apk
 22.(*).Tweetin.5.0.0.apk
 23.myapplication.flashcards_2.1.apk
 24.com.android.gpctest_2.1.11.apk
 25.JadeHS_0.5.4.apk
 26.cn.nekocode.camerasfilter_2.3.apk
 27.de.mygrades_1.2.0.apk
 28.com.openfarmanager.android_0.9.3.apk
 29.(*).com.andreadec.musicplayer_2.5.4.apk
 30.com.marlonjones.simplereminder_1.0.apk
 31.menion.android.wheremyougo_0.9.2.apk
 32.net.hyx.app.volumenotification_0.9.5.apk
 33.androidVTS_13.apk
 34.WifiLocationLogger-debug.apk
 35.uk.co.alt236.floatinginfo_2.1.apk
 36.com.example.android.wifidirect_1.0.apk
 37.com.tr.expenses_1.16.apk
 38.io.neoterm_1.2.10.apk
 39.clock.socoolby.com.clock_2017-04-20.apk
 40.kiwi.root.an2linuxclient_0.6.0.apk
 41.at.huber.youtubeDownloader_1.6.0.apk
 42.com.tomclaw.mandarin_1.7.apk
 43.in.workarounds.define_1.2.apk
 44.net.rfc1149.harassme_1.25.apk
 45.de.igloffstein.maik.aRevelation_1.0b1.apk
 46.com.saha.batchuninstaller_0.9.2.apk
 47.com.sjj.echo.umsinterface_1.3.5.apk
 48.com.netease.qa.emmagee_2.5.1.apk
 49.com.mrbimc.selinux_11.apk
 50.com.google.zxing.client.android_4.7.7.apk
 51.pt.samp.scrumCards_2.3.apk

پیوست الف

در این قسمت نام برنامه‌هایی که از مخزن F-Droid به عنوان مورد آزمون به صورت دلخواه انتخاب کرده بودیم، آمده است.

برنامه‌های انتخاب شده از

مخزن F-Droid

برنامه‌هایی که با علامت «*» مشخص شده‌اند، آسیب‌پذیر به تزریق SQL بوده‌اند. برنامه‌هایی که با علامت «**» مشخص شده‌اند، مسیر آسیب‌پذیری در آن‌ها وجود داشته است. ولی چون برنامه‌نویس از تابع پارامتری استفاده کرده است، آسیب‌پذیری قابل بهره‌برداری نیست.

1.com.makinggiants.todayhistory_0.1.0.apk
 2.com.jkenneth.droidovpn_1.0.1.apk
 3.xSocks-v1.5.0.apk
 4.com.wkovacs64.mtorch_2.1.0.apk
 5.com.pcmehanic.smarttoolkit_2.1.3.apk
 6.org.zephyrsoft.trackworktime_1.0.1.apk
 7.es.dmoral.tinylist_2017-01-14.apk
 8.com.michaeloftiadis.locationmanagerviewer_1.1.apk
 9.com.googlecode.android_scripting_6.2.0.apk
 10.net.alainindonesia.silectric_1.2.01.apk
 11.ca.frozen.rpicameraviewer_1.3.apk
 12.com.apps.adrcotfas.goodtime_1.2.0.apk
 13.org.sorz.lab.tinykeepass_0.3.2.apk
 14.dev.ldev.gpsicon_1.2.apk

- 84.org.secuso.privacyfriendlytapemeasure_1.0.apk
- 85.com.moosd.netghost_1.2.apk
- 86.com.viper.simplert_1.1.apk
- 87.com.studio4plus.homerplayer_0.7.2.apk
- 88.com.QuarkLabs.BTCeClient_2.2.apk
- 89.com.ianhanniballake.localstorage_2.2.apk
- 90.org.laurentsebag.wifitimer_1.3.0.apk
- 91.cobalt.blackberrymanager_2017-10-27.apk
- 92.*de.ub0r.android.smsdroid_1.7.2.apk
- 93.com.vlath.keyboard_2017-07-30.apk
- 94.de.visorapp.visor_1.5.0.apk
- 95.com.hollandhaptics.babyapp_2016-12-09.apk
- 96.de.pinyto.ctSESAM_1.0.apk
- 97.Paperwork-Android.apk
- 98.moe.haruae.wadb_2.4.1.apk
- 99.xyz.mustafaali.devqstiles_1.1.0.apk
- 100.com.github.shadowsocks.plugin.obfs_local_0.0.5.apk
- 101.com.afollestad.impression_0.8.0.apk
- 102.cf.vojtechh.apkmirror_3.3.4.apk
- 103.com.mmmoussa.iqra_2.2.0.apk
- 104.com.topjohnwu.magisk_5.4.0.apk
- 105.th.in.whs.k2ausbkbd_1.3.1.apk
- 106.SmartHomeMobile-1.1.2-debug.apk
- 107.org.tvbrowser.tvbrowser_0.6.4.apk
- 108.org.emunix.unipatcher_0.14.2.apk
- 109.com.sagar.screenshift2_1.1.apk
- 110.io.github.mthli.mount_1.1.1.apk
- 111.com.blabbertabber.blabbertabber_0.8.3.apk
- 112.com.cameron.spotifyadblocker_1.0.apk
- 113.busradar.madison_1.0.38.apk
- 114.com.github.albalitz.save_1.2.0.apk
- 115.com.kaiserapps.benkaiser.strettoandroidsync_1.2.2.apk
- 116.luongvo.com.todolistminimal_1.0.0.apk
- 117.expense_manager_1.1.apk
- 118.com.venomvendor.sms.deduplicate_2.0.0.apk
- 52.com.arcao.geocaching4locus_2.1.5.4.apk
- 53.mensaUpb_3.5.1.apk
- 54.com.stripe1.xmouse_2017-01-06.apk
- 55.de.arcus.playmusicexporter2_2017-05-06.apk
- 56.com.scottyab.safetynet.sample_0.3.0.apk
- 57.shinado.indi.piping_1.0.01.apk
- 58.org.ocs.android.agent_2.3.0RC.apk
- 59.cat.mvmike.minimalcalendarwidget_0.3.1.apk
- 60.com.savageorgiev.blockthis_112.apk
- 61.*net.codersgarage.iseeu_1.0.apk
- 62.Passwy_0.2.apk
- 63.com.lscore.android.scorekeeperapp_1.0.apk
- 64.eu.basicairdata.graziano.gpslogger_2017-09-30.apk
- 65.ca.rmen.android.scrumchatter_1.6.3.apk
- 66.com.stealthcotper.networktools_0.3.08.apk
- 67.org.volkszaehler.volkszaehlerapp_0.9.5.apk
- 68.com.kos.ktodo_2.1.apk
- 69.ua.naiksoftware.j2meload_1.2.5.1.apk
- 70.tk.elevenk.proxysetter_0.2.apk
- 71.com.spynet.camtest_2017-05-13.apk
- 72.de.spiritcroc.syncsettings_2.2.apk
- 73.ca.zgrs.clipper_1.2.1.apk
- 74.com.anod.appwatcher_1.9.2.apk
- 75.uk.co.ianfield.devstat_2.4.0.apk
- 76.com.mooshim.mooshimeter_2016-09-12.apk
- 77.de.florian.processlimit_1.0.apk
- 78.com.emmanuelmess.simpleaccounting_1.5.1.apk
- 79.com.halcyonwaves.apps.energize_2.2.apk
- 80.com.calintat.explorer_1.0.4.apk
- 81.org.secuso.privacyfriendlysudoku_2.1.1.apk
- 82.cipher.root.com.cipher_2.0.apk
- 83.tr.xip.timetable_1.0.apk

-
- 119.com.sketchpunk.ocomicreader_0.2.1.1.apk
 - 120.net.mm2d.dmsexplorer_0.7.9.apk
 - 121.HeadsUp-v3.1-release.apk
 - 122.open.currency_2.0.1.apk
 - 123.*com.sylwke3100.freetrackgps_0.9.apk
 - 124.com.tananaev.logcat_1.7.apk
 - 125.com.afollestad.photoaffix_2017-05-30.apk
 - 126.com.pluscubed.matlog_1.1.0.apk
 - 127.de.hannesstruss.windfish_1.2.2.apk
 - 128.github.yaa110.memento_1.1.1.apk
 - 129.com.by_syk.netupdown_1.1.3.apk
 - 130.com.vlath.beheexplorer_2.5.1.apk
 - 131.com.colorcloud.wifichat_1.0.apk
 - 132.me.varunon9.remotecontrolpc_1.4.0.apk
 - 133.com.khmelenko.lab.varis_1.00.011.apk
 - 134.com.pavelsikun.runinbackgroundpermissionsetter_1.5.apk
 - 135.**com.example.kyle.reminder_1.0.apk
 - 136.com.manoj.dlt_2.3.6.apk
 - 137.xyz.iridiumion.enlightened_31.apk
 - 138.com.github.xfalcon.vhosts_1.6.4.apk
 - 139.com.aaronjwood.portauthority.free_2.0.1.apk
 - 140.*Sieve.apk

```

setContentView(R.layout.activity_main);
setTitle("Ehsan");
Button b= (Button)
findViewById(R.id.button_submit);
EditText et= (EditText) findViewById(R.id.edit_text);
final String etStr = et.getText().toString();
b.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View view) {
setTitle(etStr);
assert(false);
}
});
}
}
}

```

برنامک شماره یک

این برنامک تنها از کلاس MainActivity تشکیل شده است:

```

public class MainActivity extends
AppCompatActivity{

@Override
protected void
onCreate(Bundle savedInstanceState){
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
Toolbar
toolbar=(Toolbar)findViewById(R.id.toolbar);
setSupportActionBar(toolbar);

FloatingActionButton
fab=(FloatingActionButton)findViewById(R.id.fab);
fab.setOnClickListener(new
View.OnClickListener()
{
@Override
public void onClick(View view){
Snackbar.make(view,"Replace with your own action",Snackbar.LENGTH_LONG).setAction("Action",null).show();
}
}
});
}
}
}

```

پیوست ب

در این بخش، کد برنامک‌هایی آمده است که خودمان برای آزمون ابزارمان، برای تشخیص آسیب‌پذیری تزریق SQL و تشخیص استثنای زمان اجرا پیاده‌سازی کرده بودیم.

برنامک‌های پیاده‌سازی خودمان

در این قسمت برنامک‌هایی که خودمان پیاده‌سازی کرده‌ایم، آمده‌اند. برای هر دو آزمون انجام شده از این ۱۰ برنامک به صورت مشابه استفاده شده‌اند. در تشخیص آسیب‌پذیری به جای استثنای زمان اجرا از پرس‌وجو به پایگاه داده استفاده شده است ولی ساختار کلی برنامک‌ها یکی بوده‌اند.

برنامک شماره صفر

در برنامک شماره صفر، تنها اجرا شدن فرایند آزمون روی SPF و با خطای زمان اجرا بررسی شده است.

```

public class MainActivity extends AppCompatActivity {

@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);

```

```

EditText
et=(EditText)findViewById(R.id.editText);
String st=et.getText().toString();

if(st.charAt(0)=='a'){
assert(false);
}

FloatingActionButton
fab=(FloatingActionButton)findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View view) {
Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
.setAction("Action", null).show();
}
});
}

@Override
Public boolean onCreateOptionsMenu(Menu menu) {
getMenuInflater().inflate(R.menu.menu_main, menu);
return true;
}

@Override
Public boolean onOptionsItemSelected(MenuItem item) {
int id=item.getItemId();

if(id==R.id.action_settings){
return true;
}

return super.onOptionsItemSelected(item);
}
}

```

```

}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
getMenuInflater().inflate(R.menu.menu_main, menu);
return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
int id=item.getItemId();

if(id==R.id.action_settings){
return true;
}

return super.onOptionsItemSelected(item);
}
}

```

برنامه شماره دو

برنامه شماره دو هم تنها از یک کلاس MainActivity تشکیل شده است. در این جا از استثنای زمان اجرا و گرفتن ورودی از کاربر استفاده شده است.

```

public class MainActivity extends AppCompatActivity {

@Override
Protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
Toolbar toolbar=(Toolbar)findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
}
}

```

```

@Override
public void onClick(View v){
if(st.charAt(0)=='a'){
assert(false);
}
}
});
}

@Override
public boolean onCreateOptionsMenu(Menu
menu){
getMenuInflater().inflate(R.menu.menu_main,me
nu);
return true;
}

@Override
public boolean
onOptionsItemSelected(MenuItem item){
int id=item.getItemId();

if(id==R.id.action_settings){
return true;
}

return super.onOptionsItemSelected(item);
}
}

```

برنامه شماره چهار

در برنامه شماره چهار، به جای خطای
 زمان اجرا، این بار آسیب پذیری تزریق SQL را
 قرار داده ایم. در اینجا خروجی پرس و جو به تابع
 نشت نمی رود.

```

public class MainActivity extends
AppCompatActivity{
@Override
protected void onCreate(Bundle
savedInstanceState){

```

برنامه شماره سه

در برنامه شماره سه، در همان کلاس
 MainActivity یک دکمه نیز قرار دارد. کاربر
 با فشردن دکمه، در صورتی که ورودی با حرف
 آغازین «a» وارد کرده باشد با استثنای
 زمان اجرا روبه رو می شود.

```

public class MainActivity extends
AppCompatActivity{

```

```

@Override
protected void onCreate(Bundle
savedInstanceState){
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
Toolbar
toolbar=(Toolbar)findViewById(R.id.toolbar);
setSupportActionBar(toolbar);

```

```

EditText
et=(EditText)findViewById(R.id.editText);
Final String st=et.getText().toString();

```

```

FloatingActionButton
fab=(FloatingActionButton)findViewById(R.id.f
ab);
fab.setOnClickListener(new View.OnClickListen
er(){

```

```

@Override
public void onClick(View view){
Snackbar.make(view,"Replacewithyourownactio
n",Snackbar.LENGTH_LONG)
.setAction("Action",null).show();
}
});

```

```

Button b=(Button)findViewById(R.id.button);
b.setOnClickListener(new View.OnClickListener()
){

```

```

}
});

}

@Override
public boolean onCreateOptionsMenu(Menu menu){
getMenuInflater().inflate(R.menu.menu_main,menu);
return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item){
int id=item.getItemId();

if(id==R.id.action_settings){
return true;
}

return super.onOptionsItemSelected(item);
}
}

```

برنامه شماره پنج

در برنامه شماره پنج، از دو Activity استفاده شده است که از طریق intent به هم مرتبط

هستند. در اینجا خطای زمان اجرا را آورده ایم.

```

public class MainActivity extends AppCompatActivity {

@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContent(R.layout.activity_main);

```

```

super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
Toolbar
toolbar=(Toolbar)findViewById(R.id.toolbar);
setSupportActionBar(toolbar);

EditText
et=(EditText)findViewById(R.id.editText);
final String st=et.getText().toString();

final SQLiteDatabase db =
openOrCreateDatabase("StudentDB",Context.MODE_PRIVATE,null);
db.execSQL("CREATE TABLE IF NOT EXISTS student(stdno VARCHAR,name VARCHAR, marks VARCHAR);");
db.execSQL("INSERT INTO student VALUES ('9031066','ehsan','20");
db.execSQL("INSERT INTO student VALUES ('9031068','hamid','19");
db.execSQL("INSERT INTO student VALUES ('9031806','seyed','20");

FloatingActionButton
fab=(FloatingActionButton)findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener(){
@Override
public void onClick(View view){
Snackbar.make(view,"Replacewithyourownaction",Snackbar.LENGTH_LONG).setAction("Action",null).show();
}
});

Button b=(Button)findViewById(R.id.button);
b.setOnClickListener(new View.OnClickListener(){
@Override
public void onClick(View v){
db.execSQL("SELECT * FROM student where stdno='"+st+"'");

```



```

getMenuInflater().inflate(R.menu.menu_main,
menu);
return true;
}

@Override
public boolean
onOptionsItemSelected(MenuItem item) {
// Handle action bar item clicks here. The action
bar will
// automatically handle clicks on the Home/Up
button, so long
// as you specify a parent activity in
AndroidManifest.xml.
int id = item.getItemId();

//noinspection SimplifiableIfStatement
if (id == R.id.action_settings) {
return true;
}

return super.onOptionsItemSelected(item);
}
}
public class SecondActivity extends
AppCompatActivity {

@Override
protected void onCreate(Bundle
savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_second);
int id = getIntent().getIntExtra("ID",0);
if(id==125) {
Log.e("TEST","ERROR FOUND!!!");
assert (false);
}
}
}
}

```

برنامک شماره شش

```

Toolbar toolbar = (Toolbar)
findViewById(R.id.toolbar);
setSupportActionBar(toolbar);

EditText et=(EditText)
findViewById(R.id.editText);
final String st =et.getText().toString();

FloatingActionButton fab =
(FloatingActionButton) findViewById(R.id.fab);
fab.setOnClickListener(new
View.OnClickListener() {
@Override
public void onClick(View view) {
Snackbar.make(view, "Replace with your own
action", Snackbar.LENGTH_LONG)
.setAction("Action", null).show();
}
});

Button b= (Button) findViewById(R.id.button);
b.setOnClickListener(new
View.OnClickListener() {
@Override
public void onClick(View v) {
if(st.charAt(0)!='a') {
Intent myIntent = new Intent(MainActivity.this,
SecondActivity.class);
myIntent.putExtra("ID", 125);
startActivity(myIntent);
}
}
});

@Override
public boolean onCreateOptionsMenu(Menu
menu) {
// Inflate the menu; this adds items to the action
bar if it is present.

```

```

if(st.charAt(0)!='a') {
Intent myIntent = new Intent(MainActivity.this,
SecondActivity.class);
myIntent.putExtra("ID", 125);
startActivity(myIntent);
}
}
});
}

```

```

@Override
public boolean onCreateOptionsMenu(Menu
menu) {
// Inflate the menu; this adds items to the action
bar if it is present.
getMenuInflater().inflate(R.menu.menu_main,
menu);
return true;
}

```

```

@Override
public boolean
onOptionsItemSelected(MenuItem item) {
// Handle action bar item clicks here. The action
bar will
// automatically handle clicks on the Home/Up
button, so long
// as you specify a parent activity in
AndroidManifest.xml.
int id = item.getItemId();

```

```

//noinspection SimplifiableIfStatement
if (id == R.id.action_settings) {
return true;
}

```

```

return super.onOptionsItemSelected(item);
}
}
public class SecondActivity extends
AppCompatActivity {
@Override

```

در برنامه شماره شش، دو Activity آمده است. در اولی اگر کاربر متنی با حرف آغازین «a» وارد کند و دکمه را فشار دهد، Activity دوم باز می‌شود و خطای زمان اجرا اتفاق می‌افتد.

```

public class MainActivity extends
AppCompatActivity {

```

```

@Override
protected void onCreate(Bundle
savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
Toolbar toolbar = (Toolbar)
findViewById(R.id.toolbar);
setSupportActionBar(toolbar);

```

```

EditText et=(EditText)
findViewById(R.id.editText);
final String st =et.getText().toString();

```

```

FloatingActionButton fab =
(FloatingActionButton) findViewById(R.id.fab);
fab.setOnClickListener(new
View.OnClickListener() {
@Override
public void onClick(View view) {
Snackbar.make(view, "Replace with your own
action", Snackbar.LENGTH_LONG)
.setAction("Action", null).show();
}
});

```

```

Button b= (Button) findViewById(R.id.button);
b.setOnClickListener(new
View.OnClickListener() {
@Override
public void onClick(View v) {

```

```
db.execSQL("INSERT INTO student
VALUES('9031068','hamid','19');");
db.execSQL("INSERT INTO student
VALUES('9031806','seyed','20');");
final TextView tv = (TextView)
findViewById(R.id.textview);
```

```
FloatingActionButton fab =
(FloatingActionButton) findViewById(R.id.fab);
fab.setOnClickListener(new
View.OnClickListener() {
@Override
public void onClick(View view) {
db.execSQL("DELETE FROM student");
Snackbar.make(view, "ALL Records Deleted!!",
Snackbar.LENGTH_LONG)
.setAction("Action", null).show();
}
});
```

```
Button b= (Button) findViewById(R.id.button);
b.setOnClickListener(new
View.OnClickListener() {
@Override
public void onClick(View v) {
final String st =et.getText().toString();
Cursor c=db.rawQuery("SELECT * FROM
student where stdno='"+st+"' ", null);
String buffer=new String();
while(c.moveToNext())
{
buffer+="stdno: "+c.getString(0)+"\n";
buffer+="Name: "+c.getString(1)+"\n";
buffer+="Marks: "+c.getString(2)+"\n\n";
}
tv.setText(buffer);
}
});
```

```
@Override
```

```
protected void onCreate(Bundle
savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_second);
int id = getIntent().getIntExtra("ID",0);
if(id==125) {
Log.e("TEST","ERROR FOUND!!!");
assert (false);
}
}
}
```

برنامک شماره هفت

برنامک شماره هفت، مانند برنامک شماره چهار
است. با این تفاوت که تابع نشت `TextView`
نیز در آن وجود دارد.

```
public class MainActivity extends
AppCompatActivity {
```

```
@Override
protected void onCreate(Bundle
savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
Toolbar toolbar = (Toolbar)
findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
```

```
final EditText et=(EditText)
findViewById(R.id.editText);
```

```
final SQLiteDatabase
db=openOrCreateDatabase("StudentDB",
Context.MODE_PRIVATE, null);
db.execSQL("CREATE TABLE IF NOT
EXISTS student(stdno VARCHAR,name
VARCHAR,marks VARCHAR);");
```

```
db.execSQL("INSERT INTO student
VALUES('9031066','ehsan','20');");
```

```

@Override
protected void onCreate(Bundle
savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
Toolbar toolbar = (Toolbar)
findViewById(R.id.toolbar);
setSupportActionBar(toolbar);

```

```

final EditText et=(EditText)
findViewById(R.id.editText);

```

```

final EditText et_del=(EditText)
findViewById(R.id.editText_del);

```

```

final SQLiteDatabase
db=openOrCreateDatabase("StudentDB",
Context.MODE_PRIVATE, null);
db.execSQL("CREATE TABLE IF NOT
EXISTS student(stdno VARCHAR,name
VARCHAR,marks VARCHAR);");

```

```

db.execSQL("INSERT INTO student
VALUES('9031066','ehsan','20');");
db.execSQL("INSERT INTO student
VALUES('9031068','hamid','19');");
db.execSQL("INSERT INTO student
VALUES('9031806','seyed','20');");

```

```

FloatingActionButton fab =
(FloatingActionButton) findViewById(R.id.fab);
fab.setOnClickListener(new
View.OnClickListener() {
@Override
public void onClick(View view) {
Snackbar.make(view, "your record with
name="+et_del.getText().toString()+" deleted!!",
Snackbar.LENGTH_LONG)
.setAction("Action", null).show();
int a=db.delete("student","name=?",new
String[]{et_del.getText().toString()});
if(a==0)

```

```

public boolean onCreateOptionsMenu(Menu
menu) {
// Inflate the menu; this adds items to the action
bar if it is present.
getMenuInflater().inflate(R.menu.menu_main,
menu);
return true;
}

```

```

@Override
public boolean
onOptionsItemSelected(MenuItem item) {
// Handle action bar item clicks here. The action
bar will
// automatically handle clicks on the Home/Up
button, so long
// as you specify a parent activity in
AndroidManifest.xml.
int id = item.getItemId();

```

```

//noinspection SimplifiableIfStatement
if (id == R.id.action_settings) {
return true;
}

```

```

return super.onOptionsItemSelected(item);
}
}

```

برنامک شماره هشت

برنامک شماره هشت مانند برنامک شماره هفت پیاده‌سازی شده است. با این تفاوت که با زدن دکمه یک Activity جدید باز می‌شود و تابع نشست آنجا وجود دارد. همچنین دکمه‌ای جدید در Activity اول وجود دارد که با فشردن آن اطلاعات یک کاربر پاک می‌شود.

```

public class MainActivity extends
AppCompatActivity {

```

```

getMenuInflater().inflate(R.menu.menu_main,
menu);
return true;
}

@Override
public boolean
onOptionsItemSelected(MenuItem item) {
// Handle action bar item clicks here. The action
bar will
// automatically handle clicks on the Home/Up
button, so long
// as you specify a parent activity in
AndroidManifest.xml.
int id = item.getItemId();

//noinspection SimplifiableIfStatement
if (id == R.id.action_settings) {
return true;
}

return super.onOptionsItemSelected(item);
}
}
public class SecondActivity extends
AppCompatActivity {

@Override
protected void onCreate(Bundle
savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_second);
int counter = 1;
String s="";
while(counter-
1<getIntent().getIntExtra("count",0)) {
s+=getIntent().getStringExtra("name"+counter)+"
\n";
s+=getIntent().getStringExtra("stdno"+counter)+"
\n";
s+=getIntent().getStringExtra("marks"+counter)+
"\n";
counter++;
}

Snackbar.make(view, "your record with
name="+et_del.getText().toString()+"cant be
deleted!!", Snackbar.LENGTH_LONG)
.setAction("Action", null).show();
else
Snackbar.make(view, "your record with
name="+et_del.getText().toString()+" deleted!!",
Snackbar.LENGTH_LONG)
.setAction("Action", null).show();
}
});

Button b= (Button) findViewById(R.id.button);
b.setOnClickListener(new
View.OnClickListener() {
@Override
public void onClick(View v) {
Cursor c=
db.query(false,"student",null,"stdno="+et.getTex
t().toString()+""",null,null,null,null,null);
Intent myIntent = new Intent(MainActivity.this,
SecondActivity.class);
int counter=1;
myIntent.putExtra("count", c.getCount());
while(c.moveToNext()) {
myIntent.putExtra("name"+counter,
c.getString(1));
myIntent.putExtra("stdno"+counter,
c.getString(0));
myIntent.putExtra("marks"+counter,
c.getString(2));
counter++;
}
startActivity(myIntent);

}
});

@Override
public boolean onCreateOptionsMenu(Menu
menu) {
// Inflate the menu; this adds items to the action
bar if it is present.

```

```
((TextView)findViewById(R.id.textView2)).setText(s);  
}  
}
```

Abstract

From Smart phone Operating Systems, Android is the most popular one. Android is open source, so we can access to codes and framework libraries. Concolic Execution used for testing softwares. This procedure has perfect code coverage and has the ability to detect bugs and vulnerabilities precisely, without false positive. Testing Android Apps is challenging due to event-driven nature and depending to SDK. In addition, unlike Java programs these apps have not any main method. Also, there is not any Android Concolic engine till now. In this work, we present a technique for detecting SQL injection vulnerability with SPF engine.

In this research, we present a heuristic method for Concolic Execution of Android Apps. This method make Concolic Execution guided and optimized. With ICFG and backward traversal of it, we push best path information to a stack. Then we use this stack in Concolic Execution for making it targeted in order to reduce testing overhead. For evaluation of our tool, we use 10 self designed and 4 Sig-Droid's testcases Apps. Results shows that our tool is faster in finding errors.

In this work, we present a method for detecting injection vulnerabilities in Android Apps with Concolic Execution and taint analysis. With static analysis, call graph and backward traversal of it, we extract desirable pathes for detection analysis. With idea of using Mock classes, we alleviate event-driven and path explosion challenges. For evaluation of our tool, we use 10 self designed and 140 F-Droid repository Apps. Overall result shows that 11 Apps are vulnerable to SQL injection.

Key Words: Concolic Execution, Guided Concolic Execution, Vulnerability Detection, Injection Vulnerability, Android Apps.



**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**

January 2018