



دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

پایان نامه کارشناسی ارشد
مهندسی رایانش امن

بهبود کارایی روش های تشخیص برنامه های اندرویدی باز بسته بندی شده

نگارش

مجتبی موذن

استاد راهنما

دکتر مرتضی امینی

بهمن ۱۴۰۱

به نام خدا
دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

پایان نامه کارشناسی ارشد

این پایان نامه به عنوان تحقق بخشی از شرایط دریافت درجه کارشناسی ارشد است.

عنوان: بهبود کارایی روش های تشخیص برنامه های اندرویدی باز بسته بندی شده

نگارش: مجتبی مودن

کمیته ممتحنین

استاد راهنما: دکتر مرتضی امینی امضاء:

استاد مشاور: استاد مشاور امضاء:

استاد مدعو: استاد ممتحن امضاء:

تاریخ:

سپاس

از استاد بزرگوارم که با کمک‌ها و راهنمایی‌های بی‌دریغشان، مرا در به سرانجام رساندن این پایان‌نامه یاری داده‌اند، تشکر و قدردانی می‌کنم. همچنین از همکاران عزیزی که با راهنمایی‌های خود در بهبود نگارش این نوشتار سهیم بوده‌اند، صمیمانه سپاسگزارم.

چکیده

با گسترش روزافزون استفاده از برنامه‌های اندرویدی در سالیان اخیر حملات موجود بر روی این سیستم عامل با افزایش قابل توجهی همراه بوده است. متن باز بودن برنامه‌های اندرویدی و در نتیجه، سهولت در دسترسی به کد منبع آن‌ها به جهت ایجاد تغییر، در کنار افزایش حملات بر روی آن‌ها، لزوم توجه به مقابله با حملات مطروحه در این زمینه را افزایش داده است. حملات بازبسته‌بندی روی برنامه‌های اندرویدی، نوعی از حملات هستند که در آن مهاجم، پس از دسترسی به کد منبع برنامه و کپی کردن آن و یا ایجاد تغییراتی که مدنظر مهاجم است، مجدداً آن را بازبسته‌بندی می‌کند. تغییر کدهای برنامه، اهداف متفاوتی نظیر تغییر کتابخانه‌های تبلیغاتی، نقض امنیت کاربر و یا ضربه به شرکت‌های تولید برنامه از تغییر گسترش برنامه‌های جعلی را دنبال می‌کند. بازبسته‌بندی برنامه‌های اندرویدی علاوه بر ماهیت تهدید کاربران و شرکت‌ها، ماهیتی پیشگیرانه نیز دارد. در این حالت توسعه‌دهندگان نرم افزار با مبهم سازی در برنامه‌های اندرویدی، سعی در پیشگیری از بازبسته‌بندی به وسیله‌ی مهاجمان دارند. تشخیص بازبسته‌بندی در برنامه‌های اندرویدی از آن جهت دارای اهمیت است که هم کاربران و هم شرکت‌های توسعه‌دهنده، می‌توانند از این موضوع ذی‌نفع باشند. تشخیص برنامه‌های بازبسته‌بندی شده، به جهت چالش‌های پیش‌رو، نظیر مبهم‌نگاری کدهای برنامه جعلی به دست مهاجم و همچنین تشخیص و جداسازی صحیح کدهای کتابخانه‌ای مسئله‌ای چالشی محسوب می‌شود. پژوهش‌های اخیر در این زمینه به صورت کلی، از روش‌های تشخیص مبتنی بر شباهت‌سنجی کدهای برنامه و یا طبقه‌بندی برنامه‌های موجود استفاده کرده‌اند. از طرفی برقراری حد واسطی میان سرعت و دقت در تشخیص برنامه‌های جعلی، چالشی است که استفاده از این دست پژوهش‌ها را در یک محیط صنعتی ناممکن ساخته است. در این پژوهش پس از استخراج کدهای برنامه به وسیله‌ی چارچوب سوت و ابزارهای دیس اسمبل، در یک روش دو مرحله‌ای کدهای برنامه‌های موجود با یکدیگر مقایسه می‌شود. پس از دیس اسمبل کدهای هر برنامه، در طی یک فرایند طبقه‌بندی مبتنی بر ویژگی‌های انتزاعی و دیداری، برنامه‌های کاندید برای هر برنامه مبدا استخراج می‌شود. سپس برای هر کلاس برنامه اندرویدی، امضایی متشکل از مهم‌ترین ویژگی‌های کدپایه از آن استخراج و پس از انجام مقایسه با کلاس‌های کتابخانه‌های اندرویدی موجود در مخزن، کتابخانه‌های اندرویدی حذف می‌شوند و در نهایت با مقایسه‌ی کدهای اصلی، برنامه بازبسته‌بندی شده مشخص می‌شود. در قسمت آزمون روش پیشنهادی در این پژوهش، توانستیم روش موجود در این زمینه را با بهبود امضای تولیدشده از هر برنامه و اضافه‌شدن مرحله‌ی پیش پردازش، سرعت تشخیص را ۴ برابر افزایش داده و در عین حال دقت روش موجود را نیز حفظ کنیم.

کلیدواژه‌ها: امنیت نرم افزار، اندروید، تشخیص بازبسته‌بندی، بهبود کارایی

فهرست مطالب

۱	مقدمه	۱
۷	مفاهیم اولیه	۲
۷	۱-۲ مبهم سازی	۷
۷	۲-۱-۱ روش های ساده	۷
۸	۲-۱-۲ روش های میانی	۸
۱۰	۲-۱-۳ روش های خاکستری	۱۰
۱۱	۲-۱-۴ روش های ترکیبی	۱۱
۱۱	۲-۱-۵ انواع مبهم نگارها	۱۱
۱۲	۲-۲ ساختار فایل های برنامه های اندرویدی	۱۲
۱۴	۲-۳ کتابخانه های اندرویدی	۱۴
۱۴	۲-۴ طبقه بندی	۱۴
۱۵	۲-۵ بازبسته بندی برنامه های اندرویدی	۱۵
۱۶	۳ تعریف مسأله و مرور کارهای پیشین	۱۶
۱۷	۳-۱ تعریف مسأله	۱۷
۱۸	۳-۲ روند کلی تشخیص برنامه های بازبسته بندی شده	۱۸
۱۸	۳-۲-۱ پیش پردازش برنامه های اندرویدی	۱۸
۱۹	۳-۲-۲ استخراج ویژگی	۱۹

۲۰	تشخیص بازبسته‌بندی ۳-۲-۳
۲۱	روش‌های تشخیص بازبسته‌بندی ۳-۳
۲۲	مبتنی بر تحلیل ایستا ۳-۳-۱
۳۶	مبتنی بر تحلیل پویا ۳-۳-۲
۳۹	سایر روش‌ها ۳-۳-۳
۳۹	پیش‌گیری از بازبسته‌بندی ۴-۳
۴۰	روش‌های مبتنی بر نشان‌گذاری ۳-۴-۱
۴۳	روش‌های مبتنی بر مبهم‌نگاری ۳-۴-۲
۴۶	مقایسه‌ی روش‌ها ۳-۵

۴ راهکار پیشنهادی ۴۸

۴۸	راهکار پیشنهادی ۴-۱
۵۰	ساخت امضای برنامه‌ک ۴-۲
۵۱	توصیف پارامترهای امضای برنامه‌ک ۴-۲-۱
۵۳	توصیف صوری امضای متد ۴-۲-۲
۵۴	توصیف صوری امضای کلاس ۴-۲-۳
۵۵	مؤلفه‌های تشخیص برنامه‌ک‌های بازبسته‌بندی شده ۴-۳
۵۵	مؤلفه‌ی تشخیص کدهای کتابخانه‌ای ۴-۳-۱
۵۹	مؤلفه‌ی یافتن نزدیک‌ترین همسایه ۴-۳-۲
۶۸	مقایسه‌ی دودویی و تشخیص برنامه‌ک‌های بازبسته‌بندی شده ۴-۳-۳

۵ ارزیابی ۷۰

۷۱	پارامترهای پژوهش ۵-۱
۷۱	مؤلفه‌ی تشخیص کتابخانه‌های اندرویدی ۵-۱-۱
۷۳	مؤلفه‌ی یافتن نزدیک‌ترین همسایه ۵-۱-۲

۷۳	۳-۱-۵ مؤلفه‌ی مقایسه‌ی دودویی و تشخیص برنامه‌های بازبسته‌بندی شده . . .
۷۳	۲-۵ مجموعه داده‌ی آزمون
۷۴	۳-۵ ارزیابی و مقایسه
۷۴	۱-۳-۵ مقایسه‌ی دودویی بدون طبقه‌بندی
۷۹	۲-۳-۵ مقایسه‌ی دودویی همراه با طبقه‌بندی
۸۰	۴-۵ تحلیل و جمع‌بندی عملکرد روش پیشنهادی
۸۲	۶ نتیجه‌گیری
۸۵	مراجع
۹۵	واژه‌نامه
۹۷	آ مطالب تکمیلی

فهرست جدول‌ها

۳۳	۱-۳ ۱۰ منبع پربازدید در پژوهش [۱]
۴۷	۲-۳ مقایسه‌ی روش‌های مبتنی بر تحلیل ایستا

فهرست شکل‌ها

- ۱-۲ نمونه‌ای از مبهم‌نگاری با استفاده از تغییر نام شناسه‌ها ۸
- ۲-۲ نمونه‌ای از مبهم‌نگاری با استفاده از قابلیت بازتاب به منظور پنهان‌سازی واسط فراخوانی شده
به نام batteryinfo ۱۰
- ۳-۲ ساختار پوشه‌ها و فایل‌های بسته‌های apk [۲] ۱۳
- ۱-۳ شمای کلی امضای متد در پژوهش ۲۱
- ۲-۳ مراحل تشخیص برنامه‌های بازبسته‌بندی شده در پژوهش آقای آلدینی [۳] ۲۷
- ۳-۳ گراف فعالیت و محتوای گره‌های آن در پژوهش نگویان [۴] ۲۹
- ۱-۴ نمای کلی راهکار پیشنهادی ۴۹
- ۲-۴ نمای کلی مؤلفه‌ی تشخیص کدهای کتابخانه‌ای ۵۶
- ۳-۴ نمای کلی مؤلفه‌ی یافتن نزدیک‌ترین همسایه ۶۱
- ۴-۴ نمونه‌ای از تعریف یک فعالیت ۶۲
- ۵-۴ نمونه‌ای از تعریف یک حق دسترسی ۶۲
- ۱-۵ مقایسه‌ی میانگین زمان اجرای روال تشخیص برنامه‌های بازبسته‌بندی شده ۷۵
- ۲-۵ مقایسه‌ی میانگین زمان اجرای مرحله‌ی تشخیص کدهای کتابخانه‌ای ۷۶
- ۳-۵ مقایسه‌ی میانگین زمان اجرای مراحل ساخت چکیده و مقایسه‌ی چکیده‌ها ۷۶
- ۴-۵ مقایسه‌ی دقت و فراخوان روش‌های چکیده‌سازی محلی در پژوهش جاری ۷۷
- ۵-۵ مقایسه‌ی میانگین زمان اجرای مراحل تشخیص پژوهش جاری با پژوهش ترکی [۵۲] ۷۷

۶-۵ مقایسه‌ی دقت و فراخوان پژوهش جاری و ترکی [۵۲] ۷۸

۷-۵ نمودار فاصله از جفت بازبسته‌بندی شده به ازای تمامی برنامه‌های بازبسته‌بندی شده . ۷۹

فصل ۱

مقدمه

سیستم عامل^۱ اندروید^۲ به دلیل سهولت در توسعه^۳ توسط توسعه دهندگان^۴ موبایلی و در نتیجه فراوانی استفاده از آن در تلفن های همراه، تلوزیون های هوشمند و دیگر دستگاه های موجود، حجم بالایی از بازار مصرفی سیستم عامل های موبایلی را به خود اختصاص داده است. بر طبق گزارش پایگاه استاتیسیتا^۵ [۵] سیستم عامل اندروید سهمی معادل ۷۱ درصدی از سیستم عامل های موبایلی را در سه ماهه ی پایانی سال ۲۰۲۲ به خود اختصاص داده است. در سال های اخیر به دلیل گسترش استفاده از این بستر^۶، فروشگاه های اندرویدی زیادی به جهت ارائه ی خدمات به کاربران به وجود آمده است. برخی از فروشگاه های رسمی مانند فروشگاه اندرویدی گوگل^۷، از ابزارهایی نظیر پلی پروتکت^۸ [۶] برای بررسی برنامه های اندرویدی موجود در فروشگاه استفاده می کنند. علاوه بر این، در سال های اخیر فروشگاه های متعدد رایگانی به وجود آمده اند که صرفاً برنامه های اندرویدی موجود در سطح وب را جمع آوری و آن را به کاربران ارائه می دهند. فروشگاه های رایگان غالباً ابزارهای مشخصی را برای حفظ امنیت کاربران استفاده نمی کنند و امنیت کاربران این دسته از فروشگاه های اندرویدی، همواره تهدید می شود. یکی از راه های مورد استفاده توسط مهاجمان برای وارد ساختن بدافزار^۹ به تلفن های همراه، بازبسته بندی نرم افزار^{۱۰} است. مطابق تعریف، بازبسته بندی شامل دانلود^{۱۱} یک برنامه، دسترسی به محتوای کدهای برنامه اصلی از طریق

Operation System^۱

Android^۲

Development^۳

Developers^۴

Statista^۵

Platform^۶

Google^۷

Play Protect^۸

Malware^۹

Software Repackaging^{۱۰}

Download^{۱۱}

روش‌های مهندسی معکوس^{۱۲} و در نهایت بازبسته‌بندی به همراه تغییر و یا بدون تغییر دادن کدهای برنامه اصلی^{۱۳} است. زبان اصلی توسعه در برنامه‌های اندرویدی، زبان جاوا^{۱۴} می‌باشد که یک زبان سطح بالا^{۱۵} محسوب می‌شود. در طی فرآیند کامپایل^{۱۶} برنامه‌های اندرویدی، مجموعه‌ی کدهای منبع در طی فرایندی به بایت‌کدهای دالویک^{۱۷} تبدیل می‌شوند و در ادامه ماشین مجازی جاوا^{۱۸}، بایت‌کدها را بر روی ماشین مقصد اجرا می‌کند [۷]. فهم و در نتیجه مهندسی معکوس زبان میانی دالویک بایت‌کدها آسان است و به همین علت موجب سهولت در بازبسته‌بندی برنامه‌های اندرویدی می‌شود.

به طور کلی بازبسته‌بندی را می‌توان از دو جهت مورد بررسی قرار داد، از دید توسعه‌دهندگان، بازبسته‌بندی شامل فرآیندی است که توسعه‌دهنده با انجام مبهم‌نگاری^{۱۹} در برنامه مورد توسعه، فهم بدنه‌ی اصلی برنامه را برای مهاجم^{۲۰} سخت می‌کند. از این دید، بازبسته‌بندی یک روش تدافعی تلقی می‌شود تا مهاجم پس از دسترسی به کد برنامه اصلی، نتواند بدنه‌ی برنامه اصلی برنامه را شناسایی و در نتیجه آن را تغییر دهد. از جهت دیگر، بازبسته‌بندی توسط فردی که برنامه متعلق به او نیست یک عمل تهاجمی محسوب می‌شود. در این حالت، مهاجم پس از دسترسی به کد برنامه اصلی، بسته به هدف او، برنامه را مجدداً بازبسته‌بندی می‌کند و آن را در فروشگاه‌های اندرویدی خصوصاً فروشگاه‌هایی که نظارت کمتری بر روی آن‌ها وجود دارد منتشر می‌کند. در دیدگاه تهاجمی، مهاجم به جهت اهدافی متفاوتی نظیر تغییر کدهای تبلیغاتی^{۲۱} در برنامه اصلی، تغییر درگاه‌های پرداخت و یا بازپخش بدافزار، اقدام به بازبسته‌بندی می‌کند. بازبسته‌بندی یکی از راه‌های محبوب مهاجمان برای انتقال بدافزارهای توسعه‌داده‌شده به تلفن همراه قربانی است [۸]. مطابق پژوهش آقای ژو و همکاران [۹] حدود ۸۵ درصد بدافزارهای موجود، از طریق بازبسته‌بندی منتشر می‌شوند. همانطور که گفته شد، برخی فروشگاه‌های اندرویدی نظیر گوگل، سازوکار مشخصی را برای تشخیص^{۲۲} بازبسته‌بندی ارائه‌داده‌اند اما بسیاری از فروشگاه‌های اندرویدی فعال و پربازدید، خصوصاً فروشگاه‌های رایگان، یا از هیچ ابزاری استفاده نمی‌کنند و یا در صورت توسعه‌ی نرم‌افزار بومی^{۲۳} خود برای شناسایی برنامه‌های بازبسته‌بندی شده، مشخصات و یا دقت آن را گزارش نکرده‌اند.

همانطور که اشاره شد، به دلیل محبوبیت و در نهایت استفاده‌ی زیاد برنامه‌های اندرویدی و همچنین نظارت کم در فروشگاه‌های مرتبط، بازبسته‌بندی یک روش پر استفاده به جهت انتقال بدافزار به تلفن همراه

^{۱۲} Reverse Engineering

^{۱۳} Orginal Application

^{۱۴} Java

^{۱۵} High Level

^{۱۶} Compile

^{۱۷} Dalvic Byte Code

^{۱۸} Java Virtual Machine

^{۱۹} Obfuscation

^{۲۰} Attacker

^{۲۱} Ad Code

^{۲۲} Detection

^{۲۳} Native

کاربران است. آقای خانمحمدی و همکاران [۱۰]، پس از بررسی برنامه‌های اندرویدی مجموعه داده‌ی^{۲۴} اندروزو^{۲۵}، دریافتند که ۵۲/۲۲٪ از برنامه‌های موجود در این مخزن توسط ویروس‌توتال^{۲۶}، بدافزار شناسایی شده‌اند. ویروس‌توتال، ابزاری متشکل از ۳۰ ضدبدافزار برای بررسی یک برنامه اندرویدی است. مطابق این پژوهش، ۷۷/۸۴٪ از برنامه‌های این مجموعه داده که بازبسته‌بندی شده‌اند، دارای نوعی از بدافزار ضدتبلیغاتی^{۲۷} بوده‌اند که موجب می‌شود تبلیغات موجود در برنامه تغییر کرده و اهداف مالی و امنیتی کاربران و توسعه‌دهندگان مخدوش شود. علاوه بر این، مطابق پژوهشی که توسط ویداس و همکاران [۱۱] انجام شده‌است، پس از پیاده‌سازی ۷ روش پربازدید به جهت تشخیص بازبسته‌بندی، در بهترین حالت، روش‌های موجود قادر به تشخیص ۷۲/۲۲٪ از برنامه‌های بازبسته‌بندی شده‌ی سه فروشگاه مطرح اندرویدی بوده‌اند. بنابراین مشخص است که تشخیص برنامه‌های بازبسته‌بندی شده، به چه میزان می‌تواند اهداف مالی و امنیتی توسعه‌دهندگان و کاربران برنامه‌ها را ارضا کند. در سال‌های اخیر ارائه‌ی یک راهکار پرسرعت به همراه دقت مناسب، همواره یکی از دغدغه‌های مهم پژوهش‌گران در این زمینه بوده‌است.

همانطور که گفته شد، بازبسته‌بندی برنامه‌های اندرویدی از دو دیدگاه تهاجمی و تدافعی قابل بررسی است. در حالتی که کاربر متقلب، برنامه اندرویدی اصلی را دچار تغییراتی می‌کند و آن را در اختیار عموم قرار می‌دهد، تشخیص بازبسته‌بندی، با استفاده از مقایسه‌ی برنامه اصلی و برنامه جعلی صورت می‌گیرد. تشخیص بازبسته‌بندی در این حالت را می‌توان به صورت کلی به دو طبقه تقسیم کرد. در حالت اول توسعه‌دهنده روش خود را مبتنی بر تحلیل برنامه مبدا و مقصد پیاده‌سازی می‌کند. عمده‌ی روش‌های موجود در این طبقه مبتنی بر تحلیل ایستا^{۲۸}ی جفت برنامه‌ها است و استفاده از تحلیل پویا^{۲۹} به جهت سرعت پایین آن، محبوبیت فراوانی ندارد [۱۲]. در سمت دیگر طبقه‌بندی^{۳۰} برنامه‌های اندرویدی وجود دارد. روش‌های موجود در این دسته، عمدتاً سرعت بالایی دارند اما در تشخیص جفت بازبسته‌بندی شده دقت پایینی را ارائه می‌دهند.

برنامه‌های اندرویدی متشکل از دو قسمت اصلی کدهای برنامه و منابع^{۳۱} هستند. کدهای برنامه، منطق^{۳۲} برنامه را تشکیل می‌دهند و رفتار برنامه با توجه به این قسمت مشخص می‌شود. از طرفی منابع برنامه، رابط کاربری^{۳۳} آن را تشکیل می‌دهند. روش‌های مبتنی بر تحلیل برنامه و یا طبقه‌بندی آن، عمدتاً

Data Set^{۲۴}
 Androzoo^{۲۵}
 Virus total^{۲۶}
 AdWare^{۲۷}
 Static^{۲۸}
 Dynamic^{۲۹}
 Classification^{۳۰}
 Resources^{۳۱}
 Logic^{۳۲}
 User Interface^{۳۳}

از ویژگی‌های موجود در منابع و یا کد آن استفاده می‌کنند. مهاجم در حالتی که می‌خواهد از محبوبیت برنامهک مبدا استفاده کند، سعی در یکسان‌سازی ظاهر برنامهک‌های مبدا و مقصد دارد به همین جهت از منابع برنامهک مبدا استفاده می‌کند و منطق برنامهک را مطابق با اهداف خود تغییر می‌دهد. در حالتی دیگر، متقلب سعی می‌کند که با استفاده از تغییر منابع برنامهک و تولید یک برنامهک تقلبی و گاهاً بدون هیچ تغییری در کد برنامهک، ادعای توسعه‌ی یک برنامهک جدید را اثبات کند. لازم به ذکر است استفاده از ویژگی‌های کدپایه^{۳۴} و منبع‌پایه^{۳۵}، به وفور در پژوهش‌های سال‌های اخیر یافت می‌شود که هر کدام معایب و مزایای خود را دارد.

در روش‌های مبتنی بر طبقه‌بندی عمدتاً تعریف تشخیص بازبسته‌بندی محدود به تشخیص دسته‌ی مشکوک که احتمال بازبسته‌بندی بودن جفت‌های داخل این دسته، بیش از سایر دسته‌ها است. تشخیص بازبسته‌بندی در این روش‌ها، محدود به تشخیص طبقه‌ی برنامهک ورودی می‌باشد و جفت بازبسته‌بندی شده مشخص نمی‌شود. از طرفی در روش‌های مبتنی بر تحلیل برنامهک، بررسی دویه‌دوی برنامهک‌های ورودی و مجموعه‌داده مدنظر است. در این روش‌ها تعریف تشخیص بازبسته‌بندی گسترش یافته و یافتن جفت بازبسته‌بندی به صورت مشخص، از اهداف اصلی پژوهش است. تغییر منابع برنامهک و همچنین مبهم‌نگاری در برنامهک بازبسته‌بندی شده، دو چالش مهم در راستای تشخیص بازبسته‌بندی است. متقلب پس از بازبسته‌بندی برنامهک، با استفاده از مبهم‌نگاری سعی می‌کند تغییرات خود و شباهت ساختار منطقی برنامهک تقلبی با برنامهک اصلی را پنهان کند. به همین جهت، تشخیص بازبسته‌بندی نیازمند ویژگی‌هایی است که مقاومت بالایی مقابل مبهم‌نگاری داشته‌باشد بدین معنا که تغییر و ایجاد ابهام در کد، به راحتی در این ویژگی‌ها قابل انجام نباشد.

در هنگام کامپایل برنامهک‌های اندرویدی، کتابخانه‌ها^{۳۶}یی که در برنامهک مورد استفاده قرار گرفته‌اند به همراه کد مورد توسعه، کامپایل شده و بایت‌کدهای دالویک آن در کنار برنامهک قرار می‌گیرد. بر اساس پژوهش آقای زیانگ و همکاران [۱۳] ۵۷٪ از کدهای برنامهک‌های مورد بررسی در این پژوهش، شامل کدهای کتابخانه‌ای بودند که دچار مبهم‌نگاری نشده‌اند. بنابراین تشخیص کدهای بازبسته‌بندی شده بدون تشخیص درست و دقیق و جداسازی کدهای کتابخانه‌ای امکان‌پذیر نیست و در صورتی که به درستی جداسازی صورت گیرد، می‌تواند نتایج منفی غلط و مثبت غلط را کاهش دهد. به صورت کلی دو روش برای تشخیص کدهای کتابخانه‌ای استفاده می‌شود، روش مبتنی بر لیست سفید^{۳۷} و روش تشخیص مبتنی بر شباهت سنجی^{۳۸}. در روش لیست سفید، لیستی از مشهورترین کتابخانه‌های موجود در مخازن کتابخانه‌ای

Code Base^{۳۴}
Resource Base^{۳۵}
Library^{۳۶}
White List^{۳۷}
Similarity^{۳۸}

اندروید نظیر ماون^{۳۹} جمع آوری می‌شود و با استفاده از نام کلاس‌ها و بسته‌های موجود، کلاس‌های کتابخانه ای تشخیص داده می‌شود. مشخص است که این روش، مقاومت بسیار کمی مقابل ساده‌ترین روش‌های مبهم‌نگاری در کتابخانه‌های اندرویدی دارد. در حالت دیگر از روش‌های مبتنی بر شباهت سنجی برای تشخیص کدهای کتابخانه‌ای استفاده می‌شود که در این روش، تحلیل ایستا روی کدهای برنامه‌ی مبدا و مخزن کتابخانه‌های اندروید صورت می‌گیرد و در نهایت از طریق شباهت‌سنجی، کدهای کتابخانه‌ای تشخیص داده می‌شوند. مشخص است که روش‌های مبتنی بر شباهت‌سنجی از دقت بیشتری، خصوصاً در صورت وجود ابهام، برخوردار هستند و تمایز بهتری میان کدهای کتابخانه‌ای و کدهای اصلی قرار می‌دهند اما اینگونه روش‌ها سرعت پایینی دارند.

پژوهش‌های ارائه‌شده در زمینه‌ی تشخیص برنامه‌های بازبسته‌بندی شده در سال‌های اخیر، عمدتاً در تلاش برای بهبود دقت و سرعت روش‌های پیشین بوده‌اند. مبهم‌نگاری باعث می‌شود که دقت روش‌های تشخیص مبتنی بر تحلیل ایستا و شباهت سنجی پایین بیاید و لزوم استفاده از ویژگی‌هایی را که مقاومت بالایی مقابل مبهم‌نگاری داشته باشند را افزایش دهد. از طرفی استفاده از ویژگی‌های مقاوم به مبهم‌نگاری، می‌تواند سرعت تشخیص را بسیار پایین آورده تا حدی که عملاً استفاده از این روش‌ها در یک محیط صنعتی را غیر ممکن سازد. در این پژوهش ما با استفاده از ترکیب روش‌های تحلیل ایستا و طبقه‌بندی منابع، به همراه شباهت‌سنجی، روشی را ارائه کرده‌ایم که در حالی که مقاومت بالایی نسبت به مبهم‌نگاری داشته‌باشد، سرعت روش‌های پیشین را نیز افزایش دهد. در این پژوهش به عنوان پیش‌پردازش، از یک طبقه‌بند^{۴۰} نزدیک‌ترین همسایه^{۴۱} برای کاهش فضای مقایسه‌ی دودویی^{۴۲} و با استفاده از ویژگی‌های مبتنی بر منبع، سرعت تشخیص بهبود داده‌شده است. با کاهش فضای مقایسه‌ی دودویی و طبقه‌بندی برنامه‌های مشکوک در یک دسته، مقایسه‌ی برنامه‌های موجود در آن دسته آغاز می‌شود. مقایسه‌ی دودویی در هر دسته مبتنی بر تحلیل ایستا و شباهت‌سنجی کدهای برنامه‌ی انجام می‌شود. ابتدا ویژگی‌هایی از هر کلاس و متد^{۴۳} در بسته‌های برنامه‌ی استخراج شده و امضا^{۴۴}ی هر کلاس ساخته می‌شود به طوری که امضای هر کلاس منحصر به فرد و تا حد امکان مختص همان کلاس باشد. نوآوری روش مطروحه، ترکیب روش‌های مبتنی بر طبقه‌بندی و روش‌های مبتنی بر تحلیل ایستا می‌باشد که در نهایت منجر به افزایش سرعت و در عین حال دقت خوب در تشخیص برنامه‌های بازبسته‌بندی شده است. حذف کدهای کتابخانه‌ای با استفاده از روشی مبتنی بر مقایسه‌ی کدهای موجود در مخزن کتابخانه‌ها و کلاس‌های برنامه‌ی انجام می‌شود. مخزن کتابخانه‌ها متشکل از ۸۷۷ کتابخانه‌ی اندرویدی جمع‌آوری شده از مخزن ماون^{۴۵} می‌باشد. در نهایت

^{۳۹}Maven Repository

^{۴۰}Classifier

^{۴۱}Nearest Neighbor

^{۴۲}Pairwise Comparison

^{۴۳}Method

^{۴۴}Signature

^{۴۵}Maven Repository

پس از تشخیص کلاس‌های کتابخانه‌های اندرویدی و حذف آن‌ها از کد برنامه، کدهای مورد توسعه به عنوان ورودی برای مقایسه‌ی دودویی و طبقه‌بندی مورد تحلیل قرار می‌گیرند. بهبود امضای کلاسی و در نهایت تولید امضای برنامه‌های اندرویدی با استفاده از ویژگی‌های مقاوم و در عین حال کوتاه و همچنین استفاده از روشی مبتنی بر طبقه‌بندی برنامه‌های اندرویدی پیش از مقایسه‌ی دودویی آن‌ها، ایده‌ی اصلی این پژوهش برای تشخیص برنامه‌های اندرویدی بازسته‌بندی شده بوده‌است.

در ادامه‌ی این نگارش، در فصل ۲ به تعریف مفاهیم اولیه مورد نیاز این پژوهش می‌پردازیم. در فصل ۳ به تعریف مسئله می‌پردازیم و همچنین مروری از کارهای پیشین را خواهیم داشت. در ادامه و در فصل ۴ روش مورد استفاده در این پژوهش، شرح داده خواهد شد و در فصل ۵ مقایسه و ارزیابی روش پیشنهادی خود را ارائه می‌دهیم. در نهایت و در فصل ۶ ضمن جمع‌بندی این گزارش علمی، به بررسی نقاط ضعف و قوت این پژوهش و همچنین ارائه‌ی پیشنهاداتی جهت بهبود آن خواهیم پرداخت.

فصل ۲

مفاهیم اولیه

در این فصل مفاهیمی را که به صورت مستقیم و غیرمستقیم در این پژوهش از آن‌ها استفاده شده است را شرح می‌دهیم. آشنایی با مفاهیم مطروحه در این فصل، منجر به درک بهتر پژوهش و راه‌حل پیشنهادی در فصل ۴ خواهد شد.

۱-۲ مبهم‌سازی

آن‌چنان که در فصل پیشین گفته شد، مبهم‌سازی را می‌توان از دو دیدگاه تهاجمی و تدافعی بررسی کرد. در این قسمت ما با توجه به هدف پژوهش که تشخیص بازبسته‌بندی به جهت دفاع می‌باشد، مبهم‌سازی را فرایندی در نظر می‌گیریم که در آن فرد مهاجم یا به بیان دیگر متقلب، برنامه‌ک اصلی را دانلود کرده و پس از دیکامپایل^۱ کردن، به نوعی تغییر می‌دهد که منطق کلی برنامه، تغییری نمی‌کند. مبهم‌سازی یکی از ارکان اصلی در فرایند بازبسته‌بندی است و هدف اصلی آن این است که ابزارهای تشخیص بازبسته‌بندی، خصوصاً در مواردی که از تحلیل ایستا استفاده می‌کنند را به اشتباه بیاندازد.

روش‌های مبهم‌سازی را از نظر میزان سختی در تشخیص به سه دسته‌ی کلی می‌توان تقسیم کرد [۱۴]:

۱-۱-۲ روش‌های ساده

راهکارهای موجود در این دسته عمدتاً بدون تغییر در برنامه‌ک اصلی رخ می‌دهد. در این روش متقلب پس از آن‌که به کدهای برنامه‌ک اصلی دسترسی پیدا کرد، آن را بدون هیچ گونه تغییری کامپایل و بسته‌بندی می‌کند.

^۱Decompile

باز بسته بندی تنها موجب تغییر در امضاء توسعه دهنده ی برنامه ک و جمع آزمون^۲ می شود. بنابراین روش هایی که مبتنی بر این دو خصوصیت هستند در این سطح دچار مشکل می شوند.

۲-۱-۲ روش های میانی

این دسته از روش های مبهم سازی، شامل روش هایی است که در آن بیشتر ویژگی های مبتنی بر معناسازی^۳ تغییر می کند و ویژگی های مبتنی بر نحو^۴ ثابت باقی می ماند. بنابراین، روش هایی که بیشتر مبتنی بر معناسازی برنامه های اندرویدی هستند، دچار خطای بیشتری در این سطح از مبهم نگاری می شوند. در ادامه به معرفی مختصری از انواع روش های مبهم نگاری مطابق با پژوهش [۱۵] در این دسته می پردازیم:

- **تغییر نام شناسه ها:** تغییر نام شناسه های موجود در برنامه شامل نام کلاس ها، متدها و یا متغیرها^۵ی موجود [۱۴]

```
1 public class a{
2     private Integer a;
3     private Float b;
4     public void a(Integer a, Float b){
5         this.a = a + Integer.valueOf(b)
6     }
7 }
```

شکل ۲-۱: نمونه ای از مبهم نگاری با استفاده از تغییر نام شناسه ها

- **تغییر نام بسته:** در این روش مبهم نگاری با استفاده از تغییر نام بسته های برنامه صورت می گیرد.
- **رمزنگاری رشته ها:** استفاده از رمزنگاری در رشته های^۷ مورد استفاده در فایل های دکس^۸، باعث کاهش سطح معناسازی^۹ و در نهایت مبهم شدن رشته ها می شود.
- **فراخوانی غیر مستقیم:** یکی از روش های ساده ی تغییر گراف فراخوانی^{۱۰}، استفاده از یک تابع واسط به عنوان تابع فراخوانده ی^{۱۱} تابع اصلی است. در این حالت تابع اولیه یک تابع واسط و تابع واسط

^۲Checksum

^۳Semantic

^۴Syntax

^۵Identifier

^۶Variable

^۷String

^۸Dex Files

^۹Semantic Level

^{۱۰}Call Graph

^{۱۱}Caller

به صورت زنجیره‌ای تابع اصلی را فراخوانی می‌کند. بدنه‌ی تابع واسط در این حالت، بسیار ساده و شامل یک فراخوانی تابع اصلی است.

- **جابه‌جایی دستورات:** جابه‌جایی دستورات موجود در برنامه‌ی اصلی، یکی از روش‌های پرکاربرد توسط ابزارهای مبهم‌نگاری است. جابه‌جایی دستورات به شکلی انجام می‌شود که استقلال هر قسمت حفظ گردد.

- **جابه‌جایی ساختار سلسله‌مراتبی:** در این روش، ساختار سلسله‌مراتبی کلاس‌های برنامه‌ی نوعی تغییر می‌کند که منطق کلاس‌ها دچار تغییر نشود.

- **ادغام و شکستن:** می‌توان توابع و یا کلاس‌های موجود در برنامه‌های اندرویدی را ادغام کرد. برای مثال می‌توان هر جایی که یک تابع صدا زده شده بود، فراخوانی تابع با بدنه‌ی تابع جایگزین شود. از طرفی می‌توان بدنه‌ی چند تابع را تحت یک تابع با یکدیگر ادغام کرد. این کار ساختار توابع فراخواننده را نیز تغییر می‌دهد. از طرفی می‌توان یک تابع را به چندین تابع مشخص شکست و بدین صورت گراف جریان برنامه‌ی را تغییر داد.

- **وارد ساختن کدهای بیهوده:** کدهای بیهوده، کدهایی هستند که اجرا می‌شوند ولی تاثیری در ادامه‌ی روند اجرایی برنامه‌ی ندارند. کدهای بیهوده عموماً دارای ساختارهای کنترلی^{۱۲} و حلقه‌های نپ^{۱۳} هستند که تاثیری در روند اجرای برنامه‌ی ندارند. ذکر این نکته حائز اهمیت است که در صورتی که در ساختار کدهای بیهوده از شروط کنترلی مبتنی بر متغیرهای پویا^{۱۴} استفاده شود آنگاه دیگر تحلیل ایستای برنامه‌های اندرویدی قادر به تشخیص این نوع از مبهم‌نگاری‌ها نیست.

- **وارد ساختن کدهای مرده:** یکی دیگر از روش‌های تغییر گراف‌های برنامه از جمله گراف جریان^{۱۵}، اضافه کردن کدهای مرده‌ای است که در ساختار گراف جریان برنامه‌های اندرویدی هیچ‌گاه اجرا نمی‌شوند اما به عنوان یک گره در گراف حضور دارند.

- **روش‌های دیگر:** روش‌های دیگری نظیر تغییر نام منابع مورد استفاده در برنامه‌های اندرویدی و حذف فایل اشکال‌زدایی^{۱۶} از روش‌های دیگری است که در این سطح به وفور مورد استفاده قرار می‌گیرد.

^{۱۲} Control's Statement

^{۱۳} Nop

^{۱۴} Dynamic Variables

^{۱۵} Flow Graph

^{۱۶} Debug File

۳-۱-۲ روش‌های خاکستری

روش‌های موجود در این دسته، مبتنی بر نحو برنامه‌های اندرویدی و خصوصاً زبان جاوا به وجود آمده‌است. عمده‌ی روش‌های مورد استفاده در این سطح، از خصوصیات مهم زبان جاوا به عنوان زبان اصلی در پیاده‌سازی برنامه‌های اندرویدی، استفاده می‌کنند. در ادامه به بررسی مهم‌ترین روش‌های موجود در این دسته می‌پردازیم.

- **بازتاب^{۱۷}**: بازتاب یکی از ویژگی‌های مهم و پیچیده‌ی زبان جاوا می‌باشد [۱۶] که امکان فراخوانی متدها و ارتباط با کلاس‌های برنامه را به صورت پویا فراهم می‌سازد. مهاجمان با استفاده از فراخوانی متدها به وسیله‌ی قابلیت بازتاب، می‌توانند نام واسط فراخوانی‌شده را پنهان سازند و بدین وسیله سطح جدیدی از مبهم‌نگاری را در برنامه‌های اندرویدی ایجاد سازند. استفاده از قابلیت بازتاب و رمزنگاری^{۱۸} رشته‌ی واسط مورد نظر، به طور کامل واسط فراخوانی‌شده را پنهان می‌کند.

```
1 Object object = new Object();
2 Method getService = Class.forName("android.os.
  ServiceManager").getMethod("getService",
  String.class);
3 Object obj = getService.invoke(object, new
  Object[]{new String("batteryinfo")});
```

شکل ۲-۲: نمونه‌ای از مبهم‌نگاری با استفاده از قابلیت بازتاب به منظور پنهان‌سازی واسط فراخوانی‌شده به نام **batteryinfo**

- **رمزنگاری دالویک بایت‌کدها**: در این روش، مهاجم در حین ساختن برنامه بازبسته‌بندی شده، قسمتی مهمی از کدهای برنامه را رمزنگاری کرده و در هنگام اجرا با استفاده از یک رویه‌ی رمزگشایی^{۱۹}، کدهای اصلی را بارگیری^{۲۰} می‌کند. این روش عمدتاً زمانی استفاده می‌شود که مهاجم نیاز به فراخوانی توابع واسط‌های برنامه‌نویسی داشته‌باشد و قسمتی را که واسط‌ها فراخوانی می‌شوند را رمزنگاری می‌کند.

- **بارگذاری پویای کلاس‌ها^{۲۱}**: زبان جاوا از قابلیت مهمی به نام بارگیری پویای کد پشتیبانی می‌کند که اجازه می‌دهد تکه کدی را که پیش از این در کد مورد توسعه‌ی یک برنامه موجود نبوده را در حین اجرا به برنامه اضافه کنیم. مهاجم با استفاده از این قابلیت زبان جاوا می‌تواند در حین اجرای

^{۱۷} Reflect
^{۱۸} Encryption
^{۱۹} Decryption
^{۲۰} Load
^{۲۱} Dynamic Class Loading

برنامک، قسمت‌هایی را به برنامه اضافه‌کند که عملاً تشخیص آن‌ها با استفاده از تحلیل‌های ایستا امکان‌پذیر نیست.

۴-۱-۲ روش‌های ترکیبی

هر ترکیبی از روش‌های گفته‌شده در سطوح مختلف را می‌توان برای مبهم‌نگاری استفاده کرد. به صورت کلی روش‌های میانی ۲-۱-۲ و روش‌های خاکستری ۳-۱-۲ را می‌توان دو دسته‌ی مهم از انواع مبهم‌نگاری به حساب آورد که به صورت گسترده در مبهم‌نگارهای رایگان و یا تجاری مورد استفاده قرار می‌گیرد.

۵-۱-۲ انواع مبهم‌نگارها

در قسمت پیشین، دریافتیم که مبهم‌نگاری، سطوح متفاوتی دارد که متقلبان برای تولید برنامک‌های بازبسته‌بندی شده از آن‌ها استفاده می‌کنند. از آنجایی که بسیاری از برنامک‌های تقلبی با استفاده از مبهم‌نگار^{۲۲}ها توسعه یافته‌اند و علاوه بر این برای ابداع یک روش مفید جهت تشخیص برنامک‌های بازبسته‌بندی شده ابتدا باید انواع مبهم‌نگارهای موجود را بررسی کرد. در پژوهشی که توسط ژانگ و همکاران [۱۷] انجام شده، ۴۳٪ از برنامک‌های بازبسته‌بندی شده‌ی مورد بررسی در این پژوهش، از مبهم‌نگاری‌های بسیار ساده‌ای نظیر تغییر نام و با استفاده از مبهم‌نگارهای رایگان، انجام شده‌است. در ادامه به بررسی چند مبهم‌نگار رایگان و تجاری^{۲۳} می‌پردازیم.

• پروگارد

پروگارد^{۲۴} یک نرم‌افزار متن‌باز رایگان به جهت بهینه‌سازی و مبهم‌نگاری در برنامه‌های جاوا مورد استفاده قرار می‌گیرد. بهینه‌سازی از طریق حذف کدهای مرده و منابع بلااستفاده انجام می‌شود و مبهم‌نگاری عمدتاً با استفاده از روش‌های مشروحه در بخش ۳-۱-۲ انجام می‌شود. [۱۸]

• آلاتوری

آلاتوری^{۲۵} یک مبهم‌نگار رایگان تولیدشده توسط شرکت روسی اسماردک^{۲۶} می‌باشد که سطوح مختلفی از مبهم‌نگاری را با توجه به فایل‌های پیکربندی^{۲۷} پوشش می‌دهد. این مبهم‌نگار از تغییرنام،

^{۲۲} Obfuscator

^{۲۳} Commercial

^{۲۴} Proguard

^{۲۵} Allatori

^{۲۶} Smardec

^{۲۷} Configuration

مبهم‌نگاری مبتنی بر تغییر گراف‌های جریان، مبهم‌نگاری فایل‌های اشکال‌زدایی و رمزنگاری داده‌های رشته‌ای پشتیبانی می‌کند. [۱۹، ۲۰]

• دکس‌گارد

این مبهم‌نگار نسخه‌ی تجاری نرم‌افزار پروگارد است که توسط شرکت گارداسکووار^{۲۸} تولید شده است. دکس‌گارد^{۲۹} را می‌توان مشهورترین و یکی از پیچیده‌ترین مبهم‌نگارهای موجود به حساب آورد. آخرین نسخه‌ی این نرم‌افزار انواع مبهم‌نگاری‌های سطح خاکستری نظیر بارگیری پویای کد و همچنین رمزنگاری کلاس‌ها و توابع را به صورت کامل انجام می‌دهد.

۲-۲ ساختار فایل‌های برنامه‌های اندرویدی

هر برنامه‌ی اندرویدی یک فایل فشرده‌شده با پسوند *apk*^{۳۰} است که به اختصار شامل چهار پوشه‌ی مهم و سه فایل می‌باشد. برای درک بهتر از روش پیشنهادی این پژوهش، در ادامه هر کدام از این قسمت‌ها را معرفی و کارکرد آن را بررسی خواهیم کرد [۲۱]. شمای کلی از ساختار برنامه‌های اندرویدی را می‌توان در شکل ۲-۳ مشاهده نمود.

• **پوشه‌ی res:** این پوشه شامل منابع برنامه‌های اندرویدی است که مربوط به رابط کاربری برنامه می‌شود. این پوشه در نهایت به فایل‌های R. نگاشت شده و هر کدام از منابع با یک شناسه^{۳۱} مشخص می‌گردد.

• **پوشه‌ی lib:** فایل‌های کامپایل‌شده‌ی بومی در این پوشه قرار می‌گیرند که شامل کتابخانه‌های اندرویدی و جاوایی نیز می‌شود. استفاده از فایل‌هایی که کامپایل شده‌اند سرعت اجرای برنامه‌های اندرویدی را بالا می‌برد لذا استفاده از آن‌ها به عنوان بسته‌های^{۳۲} از پیش آماده محبوبیت دارد.

• **فایل Classes.dex:** فایل‌های با پسوند *dex* فایل‌های دودویی^{۳۳} هستند که اطلاعات را در سطر و ستون‌های خود ذخیره می‌کنند. این فایل در برنامه‌های اندرویدی حاوی بایت‌کدهای دالویک است که توسط ماشین مجازی دالویک^{۳۴} اجرا می‌شود.

^{۲۸} Guardsquare

^{۲۹} DexGaurd

^{۳۰} Android Package

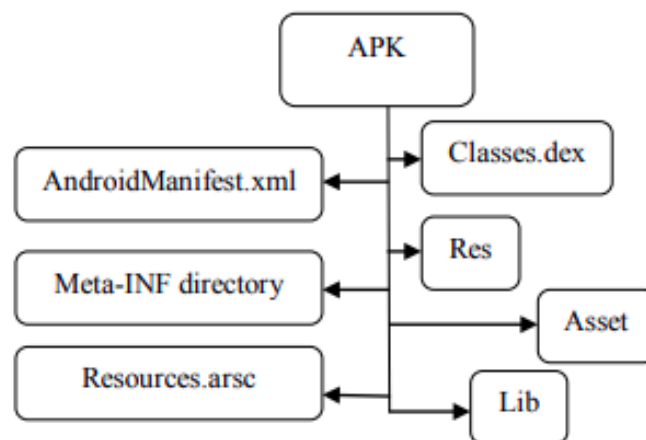
^{۳۱} Resource Id

^{۳۲} Module

^{۳۳} Binary

^{۳۴} Dalvik Virtual Machine

- فایل **AndroidManifest.xml**: پیکربندی‌های مهم فایل‌های *apk* از جمله لیست مجوزهای مورد نیاز، لیست مولفه‌ها^{۳۵} و نام بسته‌ی برنامه در این فایل نوشته می‌شود.
- پوشه‌ی **assets**: این پوشه همانند پوشه‌ی *res* برای منابع ایستا مورد استفاده قرار می‌گیرد با این تفاوت همه توسعه‌دهندگان در این پوشه می‌توانند عمق زیرپوشه‌ها را به تعداد نامتناهی افزایش دهند تا ساختار بهتری را فراهم سازند.
- پوشه‌ی **META-INF**: این پوشه شامل اطلاعات کلیدهای عمومی^{۳۶} کاربر توسعه‌دهنده‌ی برنامه است که برنامه با کلید خصوصی متناظر آن امضا شده‌است. امضای موجود در این پوشه، خاصیت صحت‌سنجی^{۳۷} دارد اما اطلاعاتی را از توسعه‌دهنده نشر نمی‌دهد و به صورت خودامضا ساخته می‌شود.
- فایل **resources.arsc**: این فایل برای انجام نگاشت^{۳۸} میان منابع موجود در پوشه‌ی *resources* و شناسه‌ی هر منبع استفاده می‌شود تا بتوان در حین اجرای برنامه‌ها، هر شناسه را به منبع آن ترجمه کرد.



شکل ۲-۳: ساختار پوشه‌ها و فایل‌های بسته‌های *apk* [۲]

^{۳۵}Component
^{۳۶}Public Key
^{۳۷}Integrity
^{۳۸}Mapping

۳-۲ کتابخانه‌های اندرویدی

کتابخانه‌های اندرویدی، بسته‌های از پیش توسعه‌یافته هستند که توسط توسعه‌دهندگان نوشته شده و توسعه‌دهندگان اندروید به جهت سهولت در پیاده‌سازی و کمک به تسریع توسعه نرم‌افزار به وفور از این نمونه‌ها استفاده می‌کنند. کتابخانه‌های اندرویدی به صورت کلی به دوبرخش کتابخانه‌های مختص برنامه‌نویسی اندرویدی و کتابخانه‌های زبان جاوا تقسیم می‌شوند. در هنگام کامپایل، تمامی کتابخانه‌هایی که توسعه‌دهنده هنگام توسعه برنامه، آن‌ها را استفاده کرده‌است به همراه کدهای مورد توسعه، کامپایل شده و در ساختار سلسله‌مراتبی تحت فایل‌های `classes.dex` قرار می‌گیرد [۲۲]. ذکر این نکته قابل توجه است که تشخیص برنامه‌های بازبسته‌بندی شده بدون شناسایی کتابخانه‌های برنامه اندرویدی مورد نظر امکان‌پذیر نیست. واضح است که در صورتی که نتوانیم کتابخانه‌های اندرویدی شاخص را از جفت برنامه‌های مورد بررسی جدا کنیم، آنگاه بخش زیادی از شباهت دو برنامه ناشی از کتابخانه‌های اندرویدی و اشتراکات موجود در آن‌ها است چرا که بسیاری از کتابخانه‌ها خصوصاً کتابخانه‌های زبان جاوا، در هر برنامه اندرویدی موجود است. از طرفی، به دلیل عدم وجود مرز مشخصی میان کدهای کتابخانه‌ای و کدهای مورد توسعه توسط توسعه‌دهندگان، شناسایی کتابخانه‌های اندرویدی تبدیل به یک چالش در زمینه تشخیص برنامه‌های بازبسته‌بندی در این حوزه شده‌است.

۴-۲ طبقه‌بندی

طبقه‌بندی اطلاعات ورودی یکی از روش‌های مرسوم در هوش مصنوعی و یادگیری ماشین است که توسط الگوریتم‌های طبقه‌بند انجام می‌شود. یک طبقه‌بند شامل مجموعه‌ای از الگوریتم^{۳۹}ها است که برای طبقه‌بندی و یا مرتب‌سازی^{۴۰} داده‌های ورودی مورد استفاده قرار می‌گیرد [۲۳]. یکی از ساده‌ترین مثال‌های موجود برای طبقه‌بندی، جداسازی هرزنامه^{۴۱}ها در سرویس‌های ایمیل است. روش‌های طبقه‌بندی نیازمند مجموعه‌ای از ویژگی‌های اطلاعات مورد بررسی به عنوان ورودی مسئله می‌باشند تا پس از اجرای الگوریتم، اطلاعات مسئله را بر اساس آن‌ها طبقه‌بندی کنند.

Algorithm^{۳۹}
Sorting^{۴۰}
Spam^{۴۱}

۵-۲ بازبسته‌بندی برنامه‌های اندرویدی

با پوشش عمیق در پژوهش‌های مرتبط با این حوزه در سالیان اخیر متوجه می‌شویم که تعاریف متنوعی برای بازبسته‌بندی در نظر گرفته شده است. برخی از پژوهش‌ها نظیر [۲۴، ۲۵] بازبسته‌بندی را در تغییر منابع و ظاهر برنامه‌ها در نظر می‌گیرند و در نهایت ویژگی‌های مبتنی بر ظاهر آن‌ها را با یکدیگر مقایسه می‌کنند. در حالی که برخی از پژوهش‌های اخیر نظیر [۲۶، ۲۷] بازبسته‌بندی را مبتنی بر تغییر ویژگی‌های کدپایه تعریف کرده‌اند. البته که نمی‌توان به هیچ کدام از تعاریف بالا خرده گرفت چرا که هر دو تعریف از نظر مهاجم و اهداف تعریف شده توسط او قابل استناد است. علاوه بر این یکی دیگر از اختلافات موجود در تعریف بازبسته‌بندی، وجود مبهم‌نگاری در برنامه‌های بازبسته‌بندی شده است. برخی از پژوهش‌ها نظیر [۲۸] بازبسته‌بندی را منوط به تغییر در امضای برنامه می‌دانند اما بسیاری از پژوهش‌های به‌روزتر، نظیر [۲۹، ۳۰] بازبسته‌بندی را تنها به تغییر منابع و یا کدهای برنامه تقلبی نسبت به برنامه اصلی می‌دانند. همانطور که مشاهده شد، هنوز تعریف مشخصی از بازبسته‌بندی در پژوهش‌ها ارائه نشده است اما به طور کلی می‌توان بازبسته‌بندی را به صورت زیر تعریف کرد:

تعریف ۱-۲ (بازبسته‌بندی) برنامه A بازبسته‌بندی یک برنامه دیگر است اگر تغییرات آن نسبت به برنامه مبدأ محدود و با حفظ کارکرد و منابع برنامه اصلی باشد.

این تعریف در این پژوهش نیز به عنوان تعریف مبنای بازبسته‌بندی در نظر گرفته شده است.

فصل ۳

تعریف مسأله و مرور کارهای پیشین

پژوهش‌های اخیر در حوزه‌ی تشخیص برنامه‌های اندرویدی بازبسته‌بندی شده نشان می‌دهد که تشخیص این دسته از برنامه‌ها تحت تاثیر دو عامل مبهم‌نگاری و جداسازی صحیح کتابخانه‌های اندرویدی قرار دارد. برخی از پژوهش‌های اخیر انجام‌شده در این حوزه، تشخیص کتابخانه‌های بسته‌ی تقلبی را با فرض عدم مبهم‌نگاری کتابخانه‌ها انجام داده‌اند که مشخصاً این فرضی نادرست است چرا که بسیاری از مبهم‌نگارهای ابتدایی نیز این کار را در کتابخانه‌های اندرویدی انجام می‌دهند. در اکثر روش‌های پیشنهادی قسمتی از روش، مختص تشخیص و جداسازی کتابخانه‌های اندرویدی است. شناسایی کدهای کتابخانه‌ای از آن جهت اهمیت دارد که تشخیص درست آن‌ها می‌تواند خطای مثبت غلط و منفی غلط را کاهش دهد. در بیشتر مواقع، خصوصاً در ابزارهای مبهم‌نگاری، متقلب هنگام بازبسته‌بندی اقدام به مبهم‌نگاری در کتابخانه‌های اندرویدی می‌کند و بدین صورت سعی در افزایش منفی غلط در ابزارهای تشخیص دارد. در صورتی که کدهای کتابخانه‌ای به درستی تشخیص و جداسازی نشوند، شباهت‌های موجود میان برنامه‌های مورد بررسی، خصوصاً در روش‌های مبتنی بر تحلیل ایستا، ناشی از کدهای کتابخانه‌ای خواهد بود. از سوی دیگر، تشخیص مبهم‌نگاری در کدهای مورد توسعه توسط متقلب، نیازمند ویژگی‌هایی از برنامه مورد نظر است که مقاومت بالایی در برابر مبهم‌نگاری داشته باشند. بدین معنا که متقلب برای تغییر این دسته از ویژگی‌ها ناچار به پرداخت هزینه‌ی زمانی و فنی باشد و در نهایت از تغییر این دست از ویژگی‌ها، پرهیز کند. در بسیاری از روش‌های ارائه‌شده در سال‌های اخیر، تشخیص برنامه‌های بازبسته‌بندی شده مبتنی بر ویژگی‌هایی صورت گرفته است که در عین مقاومت در مقابل مبهم‌نگاری، هزینه‌ی محاسباتی تشخیص برنامه‌های بازبسته‌بندی شده را افزایش می‌دهد به طوری که استفاده از این روش‌ها را عملاً در یک محیط صنعتی غیر ممکن ساخته‌است.

با توجه به اهمیت تشخیص مبهم‌نگاری و در نهایت تشخیص برنامه‌های بازبسته‌بندی شده و همچنین،

در نظر گرفتن سرعت تشخیص به عنوان یک عامل مهم، در این فصل به بررسی و مرور کارهایی می‌پردازیم که روش‌های گوناگونی را برای تشخیص برنامه‌های بازبسته‌بندی استفاده کرده‌اند و مزایا و معایب هر کدام را به صورت جدا بررسی خواهیم کرد. از آنجایی که هدف این پژوهش بهبود کارایی روش‌های تشخیص برنامه‌های بازبسته‌بندی شده است و تمرکز پژوهش بر روی تشخیص کدهای کتابخانه‌ای نبوده است، در ابتدا روند کلی تشخیص برنامه‌های بازبسته‌بندی شده را در پژوهش‌های مرتبط بیان کرده و به اختصار، روش‌های جداسازی کتابخانه‌های اندرویدی از کدهای مورد توسعه را توضیح می‌دهیم و از مرور کارهای پیشین انجام‌شده در این حوزه عبور خواهیم کرد.

در ادامه ابتدا به روند کلی تشخیص برنامه‌های بازبسته‌بندی شده می‌پردازیم و مسئله‌ی تشخیص برنامه‌های بازبسته‌بندی شده را از دیدگاه این پژوهش، شرح می‌دهیم. همچنین، دسته‌بندی انواع روش‌های تشخیص را با توجه به پژوهش‌های سال‌های اخیر بیان می‌کنیم و از هر دسته، چند پژوهش انجام‌شده را بررسی خواهیم کرد. برای درک بهتر روش پیشنهادی، در هر قسمت به بیان مزایا و معایب هر روش خواهیم پرداخت و علاوه بر این روش تشخیص کدهای کتابخانه‌ای در هر پژوهش را مشخص خواهیم کرد.

۳-۱ تعریف مسئله

علی‌رغم پژوهش‌های متعدد صورت‌گرفته در این زمینه، همانند تعریف بازبسته‌بندی، هنوز تعریف مشخصی نیز برای تشخیص بازبسته‌بندی ارائه نشده است. پژوهش‌های سال‌های اخیر در حالت کلی تشخیص بازبسته‌بندی را به دو صورت تعریف می‌کنند:

تعریف ۳-۱ (تشخیص بازبسته‌بندی مبتنی بر برنامه‌ی مبدا) / تشخیص بسته‌ی بازبسته‌بندی شده، یعنی تشخیص جفتی از برنامه‌های درون مخزن که دقیقاً جفت مشابه برنامه‌ی ورودی باشد. به بیان دیگر در این تعریف مشخص می‌شود که برنامه‌ی ورودی بازبسته‌بندی شده است یا خیر و در صورتی که بود، جفت برنامه‌ی آن درون مخزن نیز مشخص می‌شود.

تعریف ۳-۲ (تشخیص بازبسته‌بندی مبتنی بر تصمیم‌گیری برنامه‌ی مقصد) تشخیص بسته‌ی بازبسته‌بندی شده، یعنی مشخص کنیم برنامه‌ی ورودی بازبسته‌بندی شده است یا خیر. در این حالت تشخیص برنامه‌ی اصلی اهمیت‌ی ندارد و مسئله، تصمیم‌گیری^۱ درباره‌ی بازبسته‌بندی بودن یک برنامه‌ی ورودی است.

در سال‌های اخیر، اکثر پژوهش‌ها از یکی از تعاریف بالا برای تشخیص بازبسته‌بندی استفاده کرده‌اند. برای پاسخ به تعریف ۲، پژوهش‌هایی نظیر [۳۱، ۳۲، ۳۳] از روش‌های مبتنی بر مدل‌های یادگیری ماشین^۲

^۱ Decision
^۲ Machine Learning

برای تشخیص برنامه‌های بازسته‌بندی شده استفاده کرده‌اند. حال آن‌که پژوهش‌های مرتبط با تعریف ۱، نظیر [۳۴، ۳۵] بیشتر از روش‌های مقایسه‌ی دودویی و مبتنی بر شباهت‌سنجی استفاده کرده‌اند.

تعریفی که در این پژوهش مورد استفاده قرار گرفته است، تعریف ۱ است. یعنی تشخیص بازسته‌بندی منوط به تشخیص جفت برنامه اصلی در مخزن برنامه‌های موجود می‌باشد. بنابراین در طی فرایند تشخیص به دو سوال اساسی پاسخ می‌دهیم:

- آیا برنامه ورودی بازسته‌بندی شده‌ی یک برنامه‌ی دیگر است؟
- در صورتی که برنامه مورد بررسی، بازسته‌بندی شده‌ی برنامه دیگری بود، آنگاه جفت بازسته‌بندی شده‌ی برنامه ورودی کدام برنامه است.

۲-۳ روند کلی تشخیص برنامه‌های بازسته‌بندی شده

با بررسی پژوهش‌های صورت‌گرفته در حوزه‌ی تشخیص برنامه‌های بازسته‌بندی شده، درمی‌یابیم که به طور مشخص عمده‌ی این روش‌ها مراحل مشابهی را برای حل این مسئله، دنبال کرده‌اند. به طور کلی عمده‌ی روش‌های تشخیص، به عنوان ورودی، یک برنامه اندویدی شامل یک فایل با پسوند *apk* را دریافت کرده و پس از گذر از سه مرحله، مسئله را حل می‌کنند. در ادامه به بررسی این سه مرحله می‌پردازیم.

۱-۲-۳ پیش‌پردازش برنامه‌های اندرویدی

یکی از مراحل مهم در تشخیص برنامه‌های بازسته‌بندی شده، مرحله‌ی پیش‌پردازش^۳ است که تاثیر به سزایی در سرعت و دقت روش تشخیص خواهد داشت. حذف کدهای کتابخانه‌ای، حذف کدهای مرده و یا بیهوده و اعمال فیلترهای ساختاری^۴ از موارد نمونه در قسمت پیش‌پردازش است. در این قسمت روش‌های کلی مورد استفاده توسط پژوهش‌های اخیر جهت حذف کدهای کتابخانه‌ای را توضیح می‌دهیم. با توجه به مرور کارهای پیشین انجام‌شده در این حوزه، به صورت کلی دو دیدگاه در مورد تشخیص و جداسازی کتابخانه‌های اندرویدی وجود دارد:

- **مبتنی بر لیست سفید:** در این روش، لیستی از نام بسته‌های مشهور کتابخانه‌ای در برنامه‌های اندرویدی در دسترس است و با استفاده از نام بسته‌های موجود در برنامه، کدهای کتابخانه‌ای از

^۳ Pre process
^۴ Structural

کدهای مورد توسعه جدا می‌شوند. راه حل‌های مبتنی بر این روش، عموماً در مقابل مبهم‌نگاری‌های ساده‌ای نظیر تغییر نام بسته نیز مقاوم نیستند و به راحتی می‌توان آن‌ها را دور زد. مزیت این روش آن است که سرعت بالایی دارد چرا که فقط نام بسته‌ها با یکدیگر مقایسه می‌شوند اما دقت خوبی را ارائه نمی‌دهند. غالب پژوهش‌های مبتنی بر استفاده از لیست سفید، فرض کرده‌اند که تنها کدهای مورد توسعه توسط متقلب مبهم‌نگاری شده‌است و ابهام در کدهای کتابخانه‌ای را نادیده گرفته‌اند.

• **مبتنی بر شباهت‌سنجی و کدهای تکراری:** در این روش، ابتدا مخزن بزرگی از کتابخانه‌های اندرویدی تهیه می‌شود و به روش‌های گوناگون کدهای کلاسی برنامه‌ها و کدهای کتابخانه‌ای موجود در مخزن، با یکدیگر مقایسه می‌شوند و بدین طریق کتابخانه‌های اندرویدی از کدهای مورد توسعه در برنامه، جدا می‌شود. روش‌های مبتنی بر شباهت‌سنجی، بسته به این‌که از چه روشی برای یافتن کدهای تکراری استفاده می‌کنند، دقت‌های متفاوتی دارند اما به صورت کلی می‌توان گفت که مقاومت آن‌ها در مقابل مبهم‌نگاری بسیار بیشتر از روش‌های مبتنی بر لیست سفید است چرا که در صورتی که ویژگی‌های منتخب مقابل مبهم‌نگاری مقاوم باشند، آنگاه می‌توان گفت که درصد بالایی از کتابخانه‌های اندرویدی را می‌توان از کد اصلی برنامه جدا کرد.

۳-۲-۲ استخراج ویژگی

پس از حذف کدهای کتابخانه‌ای در قسمت قبلی و انجام پیش‌پردازش‌های مورد نیاز، کدهای منبع برنامه هدف، به یک طرح کلی مدل می‌شود. به صورت کلی می‌توان روش‌های تشخیص برنامه‌های بازبسته‌بندی شده را در پژوهش‌های سالیان اخیر، ناشی از تفاوت در دیدگاه در مرحله استخراج ویژگی^۵ دانست. همانطور که در شکل ۳-۱ مشاهده می‌شود، روش‌های تشخیص برنامه‌های بازبسته‌بندی به صورت کلی به دو بخش تحلیل ایستا و تحلیل پویا تقسیم می‌شود. از آنجایی که هدف ما در این پژوهش، تنها بررسی پژوهش‌هایی است که روش‌های تشخیص بازبسته‌بندی ارائه داده‌اند بنابراین روش‌هایی که توسعه‌دهندگان و شرکت‌های توسعه‌دهنده جهت جلوگیری از انجام بازبسته‌بندی پیاده‌سازی می‌کنند را به صورت خلاصه‌تر شرح خواهیم داد. به صورت کلی، می‌توان روش‌های تشخیص برنامه‌های بازبسته‌بندی شده را به دو بخش روش‌های تحلیل پویا و یا روش‌های تحلیل ایستا تقسیم کرد که در ادامه به بررسی هر کدام از این روش‌های می‌پردازیم.

• **روش‌های مبتنی بر تحلیل ایستا:** روش‌های مبتنی بر تحلیل ایستا، در مقابل مبهم‌نگاری‌های ایستا که در هنگام بازبسته‌بندی و انجام دی‌کامپایل انجام می‌شود مقاوم هستند. اما همانطور که می‌توان

^۵ Feature Extracting

حدس زد، این دسته از روش‌ها مقابل روش‌های مبهم‌نگاری همانند بازتاب مقاومتی ندارند و ممکن است دچار خطا شوند. همچنین روش‌های مبهم‌نگاری مبتنی بر رمزنگاری پویا نیز این روش‌ها را دچار خطا می‌کند. یکی از مزایای مهم روش‌های مبتنی بر تحلیل ایستا آن است که در صورت پیاده‌سازی درست و استفاده از ویژگی‌های مقاوم، می‌توانند طیف وسیعی از برنامه‌های بازبسته‌بندی شده را تشخیص دهند.

• **روش‌های مبتنی بر تحلیل پویا:** ارائه‌ی روش‌های مبتنی بر تحلیل پویا، به هدف جلوگیری از مبهم‌نگاری‌های در لحظه‌ی اجرا^۶ که در برنامه‌های اندرویدی صورت می‌گیرد، می‌باشد. به همین علت روش‌های موجود در این حوزه، عمدتاً برنامه‌ها را در هنگام اجرا بررسی و استخراج ویژگی عمدتاً در هنگام اجرا انجام می‌گیرد. به طول کلی، روش‌های مبتنی بر تحلیل پویا از مقاومت بیشتر در مقابل استفاده از راهکارهای مبهم‌نگاری برخوردار هستند. استفاده از شبیه‌سازهای جعبه‌شن^۷ به وفور در پژوهش‌های این حوزه، یافت می‌شود. یکی از چالش‌های اصلی در تشخیص برنامه‌های اندرویدی بازبسته‌بندی شده، چگونگی پیاده‌سازی شبیه‌سازها^۸ است. بسیار از شبیه‌سازها توانایی شبیه‌سازی تمامی خدمات موجود در برنامه را ندارند و برای تحلیل دقیق‌تر نیازمند استفاده از کاربران واقعی در شبیه‌سازی و استفاده از خدمات برنامه هستند. عامل دیگری که تشخیص با استفاده از تحلیل پویا را مشکل می‌کند، این است که بسیاری از بدافزارهای توسعه‌یافته، توانایی تشخیص محیط اجرای شبیه‌سازی شده را دارند و ممکن است تمامی قابلیت‌های خود و یا بخشی از آن را به جهت دور زدن سیستم‌های تشخیص پویا، پنهان کنند.

۳-۲-۳ تشخیص بازبسته‌بندی

در این مرحله با توجه به معیارها و ویژگی‌هایی که از قسمت قبل به دست آمده است و با استفاده از روش‌های گوناگون برنامه بازبسته‌بندی شده مشخص می‌شود. به صورت کلی، روش‌های پیاده‌سازی شده در این قسمت، مبتنی بر مقایسه‌ی دودویی و یا طبقه‌بندی و یادگیری ماشین هستند.

• **مقایسه‌ی دودویی:** روش‌های مبتنی بر مقایسه‌ی دودویی، مدل استخراج شده در قسمت قبلی را با استفاده از شباهت‌سنجی با برنامه‌های موجود در مخزن مقایسه می‌کند و در نهایت برنامه بازبسته‌بندی شده را مشخص می‌کند. اکثر روش‌های مبتنی بر مقایسه‌ی دودویی، جفت برنامه اصلی را نیز مشخص می‌کنند و از تعریف ۱-۳ استفاده می‌کنند بنابراین یکی از مزیت‌های این

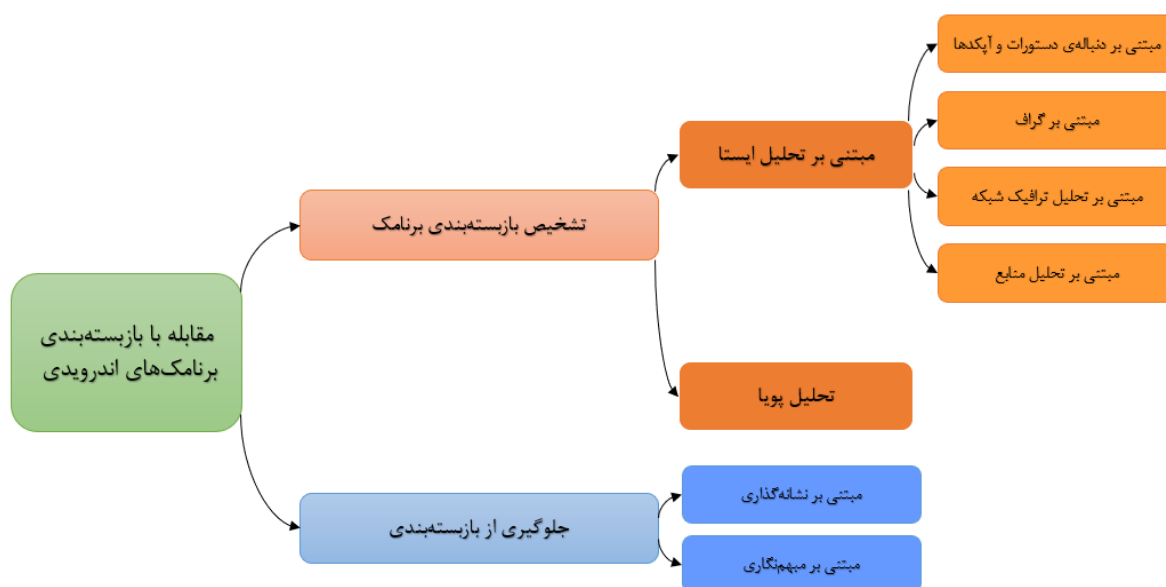
Execution Time^۶
Sand Box^۷
Simulator^۸

روش‌ها پوشش گسترده‌تر از تعریف تشخیص بازسته‌بندی است ولی در کنار آن اکثر روش‌های موجود در این زمینه، محاسبات بالایی دارند که باعث می‌شود سرعت آن‌ها کاهش یابد.

• **مبتنی بر طبقه‌بندی و یادگیری ماشین:** یکی دیگر از روش‌های تشخیص بازسته‌بندی با استفاده از ویژگی‌های مستخرج از مرحله‌ی قبل، استفاده از طبقه‌بند ها و مدل‌های یادگیری ماشین است. اکثر پژوهش‌های موجود در این زمینه از تعریف ۲-۳ برای تشخیص برنامه‌ک بازسته‌بندی شده استفاده می‌کنند. بنابراین، تنها تصمیم‌گیری در مورد بازسته‌بندی بودن یا نبودن برنامه‌ک ورودی را انجام می‌دهند. یکی از مزایای مهم این روش‌ها، سرعت بالای آن است چرا که تنها در زمان مرحله‌ی یادگیری، نیازمند محاسبات بالایی هستند و در صورتی که مدل این روش‌ها به درستی عمل کند، سرعت تشخیص به صورت قابل توجهی بالاتر از روش‌های مبتنی بر مقایسه‌ی دودویی است.

۳-۳ روش‌های تشخیص بازسته‌بندی

همانطور که در شکل ۱-۳ مشاهده می‌شود، اکثر پژوهش‌های تشخیص بازسته‌بندی از روش‌های مقایسه‌ای مبتنی بر تحلیل ایستا و پویا استفاده می‌کنند. در ادامه‌ی این قسمت ابتدا روش‌های ایستا و همچنین پژوهش‌های اخیر مرتبط با این حوزه را بررسی خواهیم کرد و در ادامه روش‌های مبتنی بر تحلیل پویا شرح داده می‌شود.



شکل ۱-۳: شمای کلی امضای متد در پژوهش

۳-۳-۱ مبتنی بر تحلیل ایستا

همانطور که گفتیم تحلیل ایستا، روشی محبوب در میان پژوهش‌های اخیر موجود در این حوزه است چرا که پیچیدگی‌های روش‌های پویا را ندارد و می‌توان به کمک آن‌ها طیف وسیعی از تشخیص مبهم‌نگاری‌ها را در برنامه‌های اندرویدی بازبسته‌بندی شده پشتیبانی کرد.

روش‌های مبتنی بر آپکد و دستورات

استفاده از آپکد^۹‌های موجود در فایل‌های دالویک، یکی از روش‌های تشخیص برنامه‌های بازبسته‌بندی شده است. هدف از پژوهش آقای ژو و همکاران [۳۶]، توسعه‌ی ابزاری به نام درویدمس^{۱۰} بوده است که توسط آن مشخص شود چه تعدادی از برنامه‌های موجود در فروشگاه‌های اندرویدی غیررسمی، بازبسته‌بندی شده‌ی برنامه‌های موجود در فروشگاه‌های رسمی هستند. همانطور که گفته شد نظارت کافی‌ای بر روی فروشگاه‌های غیر رسمی وجود ندارد، بنابراین متقلبین از این فروشگاه‌ها به عنوان یک راه امن و در دسترس برای پخش کردن برنامه‌های بازبسته‌بندی شده استفاده می‌کنند. برای استخراج امضای برنامه در این پژوهش از کدهای دالویک موجود در Classes.dex و امضای دیجیتال برنامه‌نویس در فراداده^{۱۱} استفاده شده است. پس از جداسازی کدهای کتابخانه‌ای به وسیله‌ی لیست سفید و استخراج آپکدها از فایل‌های دالویک، از یک پنجره‌ی لغزان^{۱۲} روی آپکدها استفاده شده و در نهایت چکیده^{۱۳}ی آپکدها به همراه امضای دیجیتال برنامه‌نویس، موجود در پوشه‌ی META-INF تشکیل امضای برنامه را می‌دهند. همانطور که می‌توان فهمید، فرض پژوهش این بوده است که کلید خصوصی توسعه‌دهنده لو نرفته است. در نهایت برای قسمت شباهت‌سنجی، از الگوریتم فاصله ویرایشی^{۱۴} استفاده شده است. در قسمت شباهت‌سنجی از ۲۲۹۰۶ برنامه موجود در فروشگاه‌های رسمی استفاده شده و نتایج پژوهش نشان می‌دهد که ۵ تا ۱۳ درصد از برنامه‌های موجود در فروشگاه‌های غیر رسمی، بازبسته‌بندی شده‌ی برنامه‌های فروشگاه‌های رسمی است. در پژوهش دیگری که توسط آقای ژو [۲۷] ارائه شده است، هدف پژوهش، افزایش سرعت پژوهش قبلی با استفاده از نمونه‌های n تایی از آپکدها بوده است. در این پژوهش امضای هر برنامه متشکل از قسمتی از فراداده‌ی آن شامل فایل‌های منیفست^{۱۵} و اطلاعاتی در مورد تعداد فایل‌های برنامه، توصیفات آن و چکیده‌ی آپکدهای دستورات برنامه است. این پژوهش با استفاده از یک مرحله پیش‌پردازش شامل بررسی فایل فراداده‌ی برنامه‌های موجود، فضای جست‌وجوی دودویی

Opcode^۹

DroidMoss^{۱۰}

MetaData^{۱۱}

Sliding Window^{۱۲}

Hash^{۱۳}

Edit Distance^{۱۴}

Manifest^{۱۵}

برنامک‌های مورد مقایسه را کاهش می‌دهد. دزنوز و همکاران [۳۷] روش دیگری را مبتنی بر شباهت‌سنجی روی آپکدها با استفاده از فاصله‌ی فشرده‌سازی نرمال‌شده ارائه کرده‌اند. در این پژوهش ابتدا برای هر متد با توجه به دنباله‌ی دستورات موجود امضای مشخصی تولید می‌شود و در مرحله‌ی بعد متدهایی که بکتا هستند از هر دو برنامک، بر اساس معیار فاصله‌ی فشرده‌سازی نرمال‌شده با یکدیگر مقایسه و بدین ترتیب متدهای مشابه استخراج می‌شود. در پژوهش [۳۸، ۳۹] پس از استخراج هیستوگرام‌های مربوط به تکرار آپکدها در قسمت‌های مختلف برنامک، هیستوگرام‌ها با استفاده از معیار فاصله‌ی مینی‌کاووسکی که یک معیار فاصله‌ی مبتنی بر هیستوگرام‌ها است مقایسه می‌شوند و در نهایت برنامک‌های بازبسته‌بندی شده مشخص می‌شوند. جرومه و همکاران [۴۰] در پژوهش خود با استفاده از آپکدها و تکرار آن‌ها و روش‌های مبتنی بر یادگیری ماشین برنامک‌های بازبسته‌بندی شده را تشخیص می‌دهند. در پژوهشی که توسط سرنیل و همکاران [۴۱]، ارائه شده است، از نمونه‌برداری مبتنی بر n - گرام در ۴ اندازه‌ی متفاوت ۱ تا ۴ استفاده شده است. برای شباهت‌سنجی از روش‌های طبقه‌بندی مبتنی بر درخت‌تصمیم، شبکه‌های عصبی و بردار ماشین استفاده شده است.

آقای لین و همکاران [۴۲] در پژوهش خود، با استفاده از فراخوانی‌های سیستمی^{۱۶} صدا زده شده توسط برنامک، رفتار آن را طبقه‌بندی می‌کنند. به عقیده‌ی این پژوهش، از آنجایی که اکثر بدافزارهای هم‌خانواده، در بازبسته‌بندی برنامک‌های اندرویدی، رفتار مشابه یکدیگر دارند، بنابراین استفاده از فراخوانی‌های سیستم و استخراج آن‌ها از سطح بایت‌کدهای دالویک و سطح نخ^{۱۷}، می‌تواند امضاء یکتایی از هر برنامک تولید کند. پس از استخراج بردار ویژگی^{۱۸} از فراخوانی‌های موجود با استفاده از آپکدهای برنامک، از یک طبقه‌بند بیز برای شباهت‌سنجی استفاده شده است. با توجه به روش پژوهش، شناسایی و طبقه‌بندی بدافزارهای بازبسته‌بندی شده‌ای که رفتار مشخصی ندارند و در مخزن بدافزارها موجود نیستند، یکی از ویژگی‌های مفید پژوهش ارائه شده است. فروکی و همکاران [۴۳] در پژوهش خود یک راه حل مبتنی بر استفاده از بلاک‌های ۶۴ بیتی بر روی فایل‌های دودویی برنامک‌های اندرویدی ارائه کرده‌اند. در روش ارائه شده پس از استخراج بلاک‌های ۶۴ بیتی از فایل‌ها و با استفاده از چکیده خلاصه تشابه و استخراج آنروپی^{۱۹} برای هر بلاک، بلاک‌هایی که کوچکتر و بزرگتر از یک حد کمینه و آستانه باشند حذف می‌شوند. سپس به هر بلاک با توجه آنروپی آن، یک اولویت^{۲۰} اختصاص پیدا کرده که نشان می‌دهد بلوک مورد نظر دارای آنروپی با احتمال بیشتر است. در نهایت پس از حذف بلوک‌هایی که احتمال رخداد پایین‌تری دارند نرخ مثبت غلط پژوهش کاهش یافته و از یک روش مبتنی بر بلوم‌فیلتر^{۲۱} برای مقایسه و شباهت‌سنجی استفاده می‌شود.

^{۱۶} System calls

^{۱۷} Thread

^{۱۸} Feature Vector

^{۱۹} Entropy

^{۲۰} Priority

^{۲۱} Bloom filter

آقای کو و همکاران [۴۴] از یک راه حل مبتنی بر استفاده از k -گرام برای تشخیص بسته های باز بسته بندی شده استفاده کرده اند. نویسندگان، از حذف عملوندهای^{۲۲} موجود در کدهای دودویی، به جهت کاهش مثبت های غلط در تشخیص بسته های باز بسته بندی شده استفاده کرده اند. در پژوهش کیشو و همکاران [۴۵] از یک راه حل مبتنی بر ترکیبی از دستورات کلاسی و متدهای برنامه استفاده کرده اند. در این پژوهش، در دو مرحله، ابتدا کلاس های مشابه با یکدیگر مشخص می شود و سپس در داخل کلاس های مشابه، متدهایی که یکسان هستند یافت می شود. شباهت سنجی میان کلاس ها، با استفاده از سه ویژگی، شامل لیست تمامی متدهای کلاس شامل ورودی و خروجی، لیست متغیرهای کلاسی و لیست کلاس هایی که داخل این کلاس فراخوانی شده اند، انجام می شود. پس از استخراج کلاس های مشابه، برای یافتن متدهای مشابه میان دو کلاس، از یک امضای مشترک شامل توصیف متدها به همراه نوع ورودی و خروجی آن ها و همچنین نام آن ها استفاده می شود. شباهت سنجی با استفاده از فاصله ی فشرده سازی^{۲۳} انجام شده است.

راهول و همکاران [۴۶]، روشی را پیشنهاد کرده اند که در آن استخراج ویژگی مبتنی بر درخت نحو انتزاع^{۲۴} انجام می شود. ابزار پیشنهادی در این پژوهش پس از دستیابی به کد میانی برنامه های اندرویدی و تبدیل آن به مجموعه ای از قوانین نحوی، که به صورت مجموعه ای از عبارات منظم^{۲۵} هستند، درخت نحو انتزاع را در سطح تابع تشکیل می دهد و سه ویژگی تعداد ورودی هر تابع، نام توابع صدا زده شده به صورت مستقیم و مجازی^{۲۶} و متغیرهای شرطی را استخراج می کند. سپس برای طبقه بندی از الگوریتم نزدیک ترین همسایه به جهت تشخیص باز بسته بندی استفاده شده است. نرخ منفی غلط بسیار پایین از ویژگی های مورد توجه این پژوهش است. همچنین برای ذخیره سازی درخت نحو انتزاع، از یک ساختار مبتنی بر درخت $B+$ و پایگاه داده ی *MySQL* استفاده شده است.

به صورت کلی می توان گفت که روش های مبتنی بر دستورات، خصوصاً روش هایی که به صورت مستقیم از آپکد برای تشخیص برنامه های باز بسته بنده شده استفاده می کنند، توانایی بالایی را در تشخیص برنامه های باز بسته بندی شده ارائه نمی دهند. این روش ها هم اکنون مقابل ساده ترین مبهم نگاری ها نظیر تغییر نام بسته ها و کلاس ها مقاوم نیستند و بخش زیادی از پژوهش های این حوزه، باز بسته بندی را بدون تغییر در کدهای برنامه اصلی تعریف کرده اند که به نظر با توجه به وجود مبهم نگارهای امروزی، این فرضی غلط و غیر قابل اتکا است.

^{۲۲}Operand

^{۲۳}Compression Distance

^{۲۴}Abstract Syntax Tree

^{۲۵}Regular Expressions

^{۲۶}Virtual

روش‌های مبتنی بر گراف

به صورت کلی می‌توان پژوهش‌های صورت‌گرفته در دسته روش‌های مبتنی بر گراف، را از دو جنبه بررسی کرد. دیدگاه اول پژوهش‌هایی هستند که در نهایت امضاء هر برنامه را با استفاده از یک مدل گرافی نشان می‌دهند. در این دسته از پژوهش‌ها، برای مقایسه‌ی امضاء، ناچاراً از الگوریتم‌های تشابه گراف نظیر الگوریتم‌های تشخیص گراف‌های هم‌ریخت استفاده می‌شود و به علت سربار محاسباتی بسیار بالای این پژوهش‌ها، روش‌های این دسته بسیار کند هستند. دیدگاه دوم پژوهش‌هایی هستند که صرفاً با بررسی ویژگی‌های مبتنی بر گراف‌های جریان و داده‌ای میان قسمت‌های مختلف، ویژگی‌های هر برنامه را استخراج کرده و در نهایت امضاء هر برنامه را تشکیل می‌دهند. همانطور که می‌توان حدس زد، دسته‌ی دوم از سرعت بالاتری در تشخیص برخوردار است اما چگونگی مدل‌سازی با استفاده از ویژگی‌های گرافی، بخش مهمی در پژوهش‌های این دسته است که باید به دقت پیاده‌سازی شود.

در پژوهشی که توسط آقای کروسل و همکاران [۴۷]، ابزاری مبتنی بر گراف وابستگی توسعه‌داده شده‌است. در این ابزار ابتدا، برنامه‌های موجود در مخزن با استفاده از یک ابزار شباهت‌سنجی در سطح فراداده^{۲۷}ی برنامه، به جهت افزایش سرعت، کاهش می‌یابد. پس از حذف کدهای کتابخانه‌ای به روش لیست سفید، امضاء هر برنامه با استفاده از گراف وابستگی استخراج‌شده تشکیل می‌گردد. گراف وابستگی توابع، وابستگی اجزای یک تابع از دو منظر کنترلی و داده‌ای را معرفی می‌کند. وابستگی کنترلی^{۲۸} در این پژوهش، الزام اجرای یک دستور خاص پیش از دستور دیگری است و وابستگی داده‌ای^{۲۹}، الزام مقداردهی متغیر پیش از اجرای دستور مرتبط با آن است. در قسمت شباهت‌سنجی پس از ساخت گراف وابستگی داده‌ای، با استفاده از الگوریتم تشخیص گراف‌های هم‌ریخت VF2^{۳۰} شباهت‌سنجی انجام شده و بسته‌های بازبسته‌بندی شده مشخص می‌شوند. در پژوهش دیگری که توسط آقای سان [۹] انجام شده‌است، هدف پژوهش افزایش دقت تشخیص برنامه‌های بازبسته‌بندی شده با تاکید بر شبیه‌سازی رفتار برنامه بوده‌است. در این پژوهش، واسطه‌های برنامه‌نویسی برنامه‌های اندرویدی مشخص کننده‌ی رفتار اصلی برنامه در نظر گرفته شده‌است. برای ساخت گراف هر برنامه، از گراف جریان مبتنی بر فراخوانی واسطه‌های اندرویدی استفاده شده و در نهایت پس از استخراج گراف، هر گراف نمایانگر امضاء یک برنامه می‌باشد. در گراف حاصل هر گره گراف حاوی اطلاعات یک واسطه و یال‌های گراف شامل جریان کنترلی بین واسطه‌های اندرویدی است. برای شباهت‌سنجی، از الگوریتم VF2 جهت تشخیص گراف‌های هم‌ریخت استفاده شده‌است. در مرحله‌ی آخر برنامه‌هایی که امضاء مشابه و یکسانی در تشخیص هم‌ریختی دریافت کرده‌اند به عنوان برنامه‌های بازبسته‌بندی شده در نظر گرفته می‌شوند.

^{۲۷} Meta Data

^{۲۸} Control Dependency

^{۲۹} Data Dependency

^{۳۰} Graph Isomorphism

پژوهش آقای هو و همکاران [۴۸] شامل دو مرحله‌ی ساخت گراف فراخوانی متدهای^{۳۱} برنامه و ماژول تشخیص بازبسته‌بندی است. پس از دیس‌اسمیل کردن فایل‌های برنامه، گراف فراخوانی متدهای برنامه تشکیل شده و تشکیل جنگلی از گراف‌های متصل و جدا از هم می‌دهند. سپس با استفاده از فراخوانی واسط‌های اندرویدی موجود در هر متد، گراف به دو بخش فراخوانی‌های حساس^{۳۲} و غیرحساس^{۳۳} تقسیم می‌شود و با توجه به میزان حساسیت واسط‌های فراخوانی شده، امتیاز اولویت^{۳۴} به هر گراف نگاشت می‌شود و در نهایت با استفاده از مقایسه‌ی گرافی مبتنی بر امتیاز اولویت، شباهت‌سنجی انجام می‌شود.

از آنجایی که پژوهش‌های مبتنی بر گراف در تشخیص برنامه‌های بازبسته‌بندی، عمدتاً به دلیل استفاده از روش‌های تشخیص گراف‌های هم‌ریخت‌کننده هستند، ژو و همکاران [۴۹] روشی برای افزایش سرعت در تشخیص ارائه کرده‌اند. در این پژوهش در ابتدا ماژول‌های اصلی برنامه که رفتار اصلی آن را شکل می‌دهند شناسایی می‌شود. برای شناسایی ماژول‌های اصلی برنامه، از یک گراف جهت‌دار مبتنی بر ارتباط بسته^{۳۵}‌های برنامه با یکدیگر استفاده شده و در نهایت یال‌های گراف بر اساس میزان ارتباطات بین بسته‌ها، مقداردهی می‌شود. با تشکیل گراف وزن‌دار ابتدایی، بسته‌هایی که ارتباط آن‌ها بر اساس وزن یال بین دو بسته از یک مقدار آستانه بیشتر باشد، با یکدیگر ادغام می‌شوند و رویه‌ی ادغام بسته‌ها در یک روند بازگشتی تا زمانی که هیچ بسته‌ای را نتوان با یکدیگر ادغام کرد تکرار می‌شود. در این حالت بسته‌ی نهایی شامل بسته‌ی اصلی برنامه است که منطق برنامه در این قسمت پیاده‌سازی شده است. رویه‌ی ساخت گراف ارتباطی بین بسته‌ها و ایده‌ی استفاده شده در قسمت ادغام بسته‌های اصلی با یکدیگر، ایده‌ای نو در این حوزه است که منجر به افزایش سرعت تشخیص نسبت به تمامی روش‌های گرافی شده است. برای مقایسه‌ی میان ماژول‌های اصلی برنامه، ابتدا اصلی‌ترین ماژول شامل بیشترین تعداد فعالیت، مشخص می‌شود و مقایسه میان ماژول‌های اصلی برنامه‌های اندرویدی، با استفاده از یک بردار ویژگی متشکل از فراخوانی واسط‌ها و مجوزهای^{۳۶} درخواستی انجام می‌شود. برای مقایسه از درخت VP ^{۳۷} استفاده شده است که منجر به افزایش سرعت در کنار دقت مناسب شده است. برخلاف روش‌های معمول گرافی و در نهایت مقایسه‌ی دودویی، روش پیاده‌سازی شده در پژوهش ژو، با مرتبه‌ی زمانی $n \cdot \log n$ یکی از پر سرعت‌ترین روش‌های مبتنی بر تشکیل گراف می‌باشد.

در پژوهش دیگری که به جهت افزایش سرعت در دسته پژوهش‌های مبتنی بر گراف ارائه شده است، چن و همکاران [۵۰] با استفاده از مدل کردن گراف به یک فضای سه‌بعدی، سرعت تشخیص و مقایسه‌ی

^{۳۱}Method Invocation Graph

^{۳۲}Sensitive

^{۳۳}Non-Sensitive

^{۳۴}Priority Score

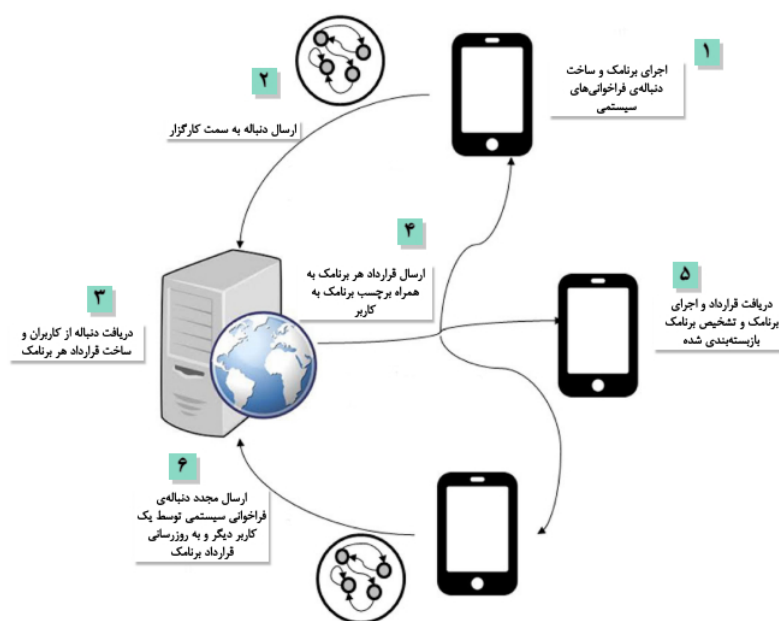
^{۳۵}Package

^{۳۶}Premissions

^{۳۷}Vantage-Point Tree

گراف‌های هم‌ریخت را به مراتب افزایش داده‌اند. ایجاد کدهای مرده^{۳۸} در کدهای برنامه، منجر به تغییر گراف جریان برنامه‌های اندرویدی می‌شود به همین جهت در این پژوهش تمامی گره‌های گرافی که نشان‌دهنده متدهای برنامه هستند به یک فضای سه‌بعدی نگاشت شده و مرکز جرم هر گراف با توجه به مختصات گره‌های گرافی تعیین می‌شود. در قسمت مقایسه‌ی گرافی، مرکز جرم گراف‌های متناظر با یکدیگر مقایسه شده و کاندیدهای بازبسته‌بندی مشخص می‌شود. در مرحله‌ی بعد برای مقایسه‌ی گره‌های هر گراف و تطبیق گراف‌های کاملاً متناظر، از مقایسه‌ی فاصله‌ی ویرایشی گره‌های متناظر استفاده می‌شود. روش ارائه‌شده علاوه بر مقاومت بالا مقابل مبهم‌نگاری، ناشی از مدل‌کردن برنامه در یک فضای گرافی، به دلیل استفاده از روشی نو در مقایسه‌ی گراف‌های مخزن برنامه‌ها، سرعت بسیار بالاتری از روش‌های پیشین دارد. برای حذف کتابخانه‌های اندرویدی از روش لیست سفید مبتنی بر اندازه‌ی بسته‌های مورد مقایسه استفاده شده‌است.

بر خلاف پژوهش‌های رایج در حوزه‌ی تشخیص بازبسته‌بندی برنامه‌های اندرویدی، در پژوهش آقای آلدینی و همکاران [۳]، از یک معماری کارخواه-کارگزار^{۳۹} مطابق با شکل ۲-۳، برای تشخیص برنامه‌های اندرویدی بازبسته‌بندی شده استفاده شده‌است. در این معماری یک برنامه بر روی دستگاه اندرویدی کاربران نصب می‌شود و شروع به ثبت و ارسال فراخوانی‌های سیستمی به کارگزار می‌کند.



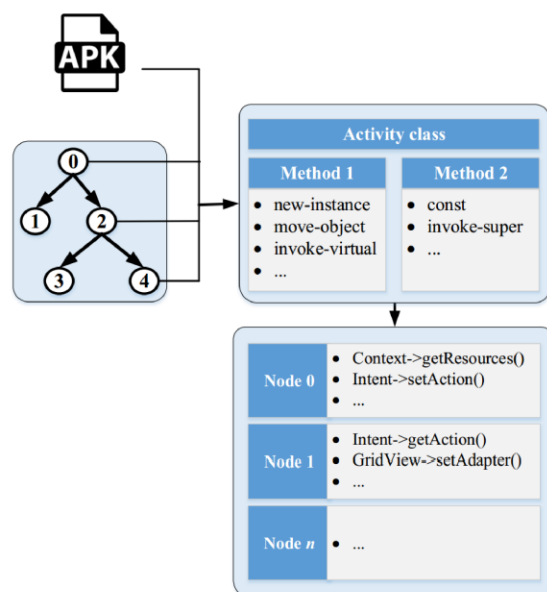
شکل ۲-۳: مراحل تشخیص برنامه‌های بازبسته‌بندی شده در پژوهش آقای آلدینی [۳]

ایده‌ی پژوهش، استفاده از فراخوانی‌های سیستمی برای شبیه‌سازی ایستای رفتار برنامه‌های اندرویدی

بوده است. اثرانگشت برنامه‌ک توسط گراف فراخوانی‌های سیستمی ارسالی از سمت کاربران در سمت کارگزار، تشکیل می‌شود سپس با استفاده از اثرانگشت موجود، یک مدل برنامه‌ک تحت عنوان قرارداد ساخته شده و این مدل به برنامه‌ی کارخواه فرستاده می‌شود. در سمت کارخواه، برنامه‌ک نصب‌شده در مرحله‌ی اول، فراخوانی‌های سیستمی برنامه‌ک موجود را با قرارداد فرستاده‌شده مطابقت می‌دهد و در صورتی که فاکتورهای نظیر نوع و تعداد فراخوانی‌های اندرویدی برنامه‌ک، با مدل پیش‌بینی شده یکسان نباشد، هشدار لازم از طریق برنامه‌ک نصب‌شده روی کارخواه به کاربر داده می‌شود. در کنار استفاده از گراف فراخوانی‌های سیستمی، پیاده‌سازی یک معماری کارخواه و کارگزار یکی از ایده‌های نو در این پژوهش بوده است. همچنین به دلیل استفاده از این معماری، پردازش سمت کارخواه به حداقل خود رسیده است و تحلیل برنامه‌ک نیازمند هیچ دانش اولیه‌ای از سمت کارخواه نمی‌باشد. علاوه بر این، تعداد برنامه‌ک‌های موجود در مخزن کارگزار، به صورت مرتب افزایش پیدا کرده و این موجب پویایی مخزن برنامه‌ک‌های اندرویدی پژوهش می‌شود.

پژوهش دیگری در زمینه‌ی روش‌های تشخیص بازبسته‌بندی مبتنی بر گراف توسط جنگ و همکاران [۵۱] ارائه شده است. در این پژوهش، پس از استخراج گراف ارتباط بین کلاس‌ها از داخل بایت‌کدهای دالویک برنامه‌ک اندرویدی، امضای هر برنامه‌ک شامل یک بردار ویژگی، متشکل از فراخوانی‌های کلاسی برنامه‌ک می‌باشد. مقایسه‌ی بردارهای برنامه‌ک‌های موجود به وسیله‌ی الگوریتم یافتن بزرگ‌ترین زیردنباله‌ی مشترک^{۴۰} انجام شده و تعیین یک حد آستانه در زیردنباله‌های مشترک، برنامه‌ک بازبسته‌بندی شده شناسایی می‌شود. روش مورد نظر را می‌توان به نوعی یک روش مبتنی بر گراف و دنباله‌ی آپکدهای برنامه‌ک توصیف کرد چرا که از هر دو ایده‌ی مدنظر استفاده نموده است. یکی دیگر از آخرین پژوهش‌های موجود در این دسته، در سال ۲۰۲۱ توسط نگویان [۴] مطرح شده است. پژوهش مورد نظر مبتنی بر استخراج گراف فعالیت^{۴۱} در برنامه‌ک‌های اندرویدی است. همانطور که در شکل ۳-۳ مشاهده می‌شود، گره‌های گراف مورد نظر شامل لیستی از واسطه‌های فراخوانی‌شده در آن فعالیت و یال‌های گراف، نشان‌دهنده‌ی یک انتقال از یک فعالیت به فعالیت دیگر است. یکی از معایب این پژوهش استفاده از الگوریتم VF2 به عنوان الگوریتم اصلی برای مقایسه و یافتن گراف‌های هم‌ریخت است که باعث کاهش سرعت پژوهش شده است.

^{۴۰} Longest common Subsequence
^{۴۱} Activity Graph



شکل ۳-۳: گراف فعالیت و محتوای گره‌های آن در پژوهش نگویان [۴]

به عنوان آخرین پژوهش صورت گرفته در این قسمت، پژوهش [۵۲، ۵۳] را بررسی خواهیم کرد. پژوهش [۵۳] که الهام‌دهنده‌ی پژوهش [۵۲] می‌باشد با استفاده از پیمایش گرافی بر روی گراف جریان برنامه‌های اندرویدی و استخراج ویژگی‌های متعدد نظیر واسطه‌های اندرویدی، فراخوانی توابع و استفاده از فیلترهای ساختاری نظیر اندازه‌ی طول امضاء، امضای هر کلاس را تشکیل می‌دهد و در نهایت با استفاده از الحاق تمامی کلاس‌های برنامه (مرتب شده به صورت الفبایی) امضاء مخصوص هر برنامه ساخته می‌شود. تفاوت این دو پژوهش بیشتر در ساختار امضاء هر برنامه می‌باشد که در فصل ۴ به توصیف بیشتر این دو پژوهش و شرح تفاوت‌های آن خواهیم پرداخت.

به طور کلی می‌توان گفت که روش‌های گرافی تشخیص برنامه‌های بازسته‌بندی شده از دقت بالایی در تشخیص برخوردار هستند. از طرفی اکثر روش‌های ارائه‌شده در این دسته، خصوصاً آن‌هایی که در نهایت هر برنامه را به یک طرح گرافی مدل می‌کنند، روش‌های تشخیص گراف‌های هم‌ریخت را در قسمت مقایسه به کار گرفته‌اند که این موضوع باعث می‌شود تا سربار محاسباتی سنگینی به پژوهش‌های مطرح وارد و سرعت تشخیص را کند سازد. علاوه بر این همانطور که بررسی شد روش‌های ارائه‌شده، در صورتی که مدل‌های گرافی را در فضای دیگری بررسی کنند، سرعت تشخیص روش بالاتر رفته و می‌توان از دقت در تشخیص مسائل گرافی نیز استفاده نمود.

روش های مبتنی بر تحلیل ترافیک شبکه

از آن جایی که ترافیک شبکه‌ی^{۴۲} عبوری از برنامه‌هایی که مبتنی بر فضای مجازی عمل می‌کنند، می‌تواند رفتار خوبی از برنامه را مدل کند بنابراین استفاده از ویژگی‌های مبتنی بر این ترافیک شبکه، از محبوبیت بالایی میان توسعه‌دهندگان، برخوردار است. به عنوان اولین پژوهش مورد بررسی، پژوهش ارائه‌شده توسط وو و همکاران [۵۴] را بررسی خواهیم کرد. در این پژوهش امضای هر برنامه مبتنی بر ترافیک http تولیدشده توسط آن درست می‌شود. در مرحله‌ی اول این پژوهش، تمامی ترافیک تولیدشده توسط برنامه جمع‌آوری شده و در مرحله‌ی بعدی ترافیک http جداسازی و تحلیل روی این دسته ادامه پیدا می‌کند. ترافیک حاصل از برنامه‌های اندرویدی در این پژوهش به دو دسته‌ی کلی تقسیم می‌شود:

- ترافیک مرجع^{۴۳}: ترافیک تولیدشده توسط برنامه توسعه‌یافته

- ترافیک کتابخانه‌ای: ترافیک تولیدشده توسط کدهای کتابخانه‌ای

جهت جداسازی ترافیک مرجع و ترافیک کدهای کتابخانه‌ای از الگوریتم‌های تطبیق جریان ترافیک^{۴۴} http و الگوریتم تطبیق هانگین^{۴۵} استفاده شده‌است. در قسمت شباهت‌سنجی از درخت جست‌وجوی VPT به جهت افزایش سرعت به دلیل متوازن بودن درخت، استفاده شده است. پژوهش مورد نظر ایده‌ای نو در زمینه‌ی تشخیص برنامه‌های بازبسته‌بندی شده است اما مشکل اصلی این پژوهش آن است که در مقابل ترافیک رمز نشده مقاوم نیست و عملاً در برنامه‌هایی که تمامی ترافیک تولیدی توسط آن‌ها رمزگذاری شده‌است ناکارآمد می‌شود.

در پژوهشی که توسط شهری^{۵۵} [۵۵] ارائه‌شده است تفکیک ترافیک به وسیله‌ی یک طبقه‌بند انجام می‌شود. پس از تفکیک ترافیک متغیرهای هر بسته شامل Request, Value, Get, Host مشخص می‌شود. ترافیک‌های از نوع Request در این قسمت به دو نوع اجباری و یا غیراجباری تقسیم می‌شود. ترافیک غیر اجباری شامل ترافیک مصرفی برنامه برای ارتباط با واسط‌های دسترسی عنوان شده‌است. جداسازی ترافیک‌های شبکه از آن جهت اهمیت دارد که بسیاری از کتابخانه‌های رایگان و یا حتی بدافزارهای موجود، یک نقطه‌ی دسترسی^{۴۶} توسط واسط‌های برنامه‌نویسی برای دسترسی به کتابخانه‌ها ایجاد کرده‌اند که برنامه‌های اندرویدی به وفور از این نفاط دسترسی استفاده می‌کنند. از طرفی تشخیص و جداسازی ترافیک اجباری و غیر اجباری موجب کاهش خطای منفی غلط می‌شود. برای شباهت‌سنجی ترافیک اجباری شامل

^{۴۲} Network Traffic

^{۴۳} Source

^{۴۴} Traffic Stream Matching

^{۴۵} Hungarian Matching

^{۴۶} EndPoint

ترافیک اصلی برنامه، از یک الگوریتم جریان درخواست^{۴۷} مبتنی بر فاصله‌ی اقلیدوسی^{۴۸} استفاده شده است. پیشنهاد جداسازی ترافیک کتابخانه‌ای و اصلی به وسیله‌ی نقطه‌های دسترسی واسطه‌های برنامه‌نویسی ایده‌ای نو در این پژوهش است که هر چند کامل نمی‌تواند ترافیک شبکه را جداسازی کند اما در صورت تکامل می‌تواند دقیق‌تر عمل کند.

در پژوهش دیگری که توسط آقای هه^[۵۶] ارائه شده، از یک طبقه‌بند به جهت تشخیص برنامه‌های بازبسته‌بندی شده استفاده شده است. در پژوهش اخیر، ابتدا تمامی ترافیک کاربران متصل به یک شبکه به یک کارگزار سطح فرستاده می‌شود و این کارگزار ترافیک شبکه را برچسب‌زنی^{۴۹} کرده و پس از استخراج ویژگی‌هایی نظیر محتوای^{۵۰} اطلاعات هر بسته، آن‌ها را برای محاسبه‌ی امضا، سمت یک کارگزار مرکزی که به صورت ابری^{۵۱} خدمات ارائه می‌کند، می‌فرستد. پس از ارسال ویژگی‌های مستخرج به سمت کارگزار ابری، فرایند شباهت‌سنجی آغاز می‌گردد. به جهت حفظ کامل حریم خصوصی کاربران، تحلیل ترافیک تنها بر روی ترافیک رمزگذاری نشده‌ی (http) انجام می‌شود. در سمت کارگزار، با استفاده از تحلیل ترافیک شبکه‌ی هر کاربر، جریان اطلاعات در ترافیک برچسب‌زده شده مشخص می‌شود و سپس با حذف قسمتی از ترافیک در هر جریان به جهت کاهش اندازه، مانند پاسخ درخواست‌ها^{۵۲} و قسمتی از سربرگ^{۵۳} ترافیک، طبقه‌بندی صورت می‌گیرد. برای طبقه‌بندی از الگوریتم پرسرعت پژوهش^[۵۷] استفاده شده است.

مالک و همکاران^[۵۸] در پژوهش دیگری تحت عنوان CREDROID از یک روش مبتنی بر درخواست نام دامنه^{۵۴} برای تشخیص برنامه‌های بازبسته‌بندی شده استفاده کرده است. در این پژوهش ابتدا ترافیک کاربر برای تحلیل به یک کارگزار مورد اعتماد فرستاده شده و فرایند بررسی عمیق ترافیک مورد نظر آغاز می‌شود. در ادامه و در سمت کارگزار مورد اعتماد، ترافیک کاربر ارزیابی شده و درخواست‌های نام دامنه جداسازی و برای هر کاربر برچسب‌گذاری می‌شود. در قسمت شباهت‌سنجی ترافیک درخواستی، از یک روش مبتنی بر فاصله‌ی اقلیدوسی جهت بررسی متن درخواست‌های دامنه استفاده می‌شود و در نهایت برنامه‌های بازبسته‌بندی شده شناسایی می‌شوند.

از آنجایی که بازبسته‌بندی برنامه‌های اندرویدی روشی محبوب برای حمله‌کنندگان به جهت تزریق و گسترش بدافزارهای اندرویدی است، تمرکز پژوهش ایلند و همکاران^[۵۹] بر روی برنامه‌های اندرویدی بازبسته‌بندی شده و حاوی ترافیک مشکوک به بدافزار، با توجه رفتار مدل‌شده بر اساس درخواست‌های دامنه می‌باشد. این روش، مبتنی بر ترافیک حاصل از درخواست نام دامنه توسط برنامه‌های بازبسته‌بندی شده و

Request Flow^{۴۷}

Euclidian Method^{۴۸}

Labeling^{۴۹}

Packet Content^{۵۰}

Cloud^{۵۱}

Response^{۵۲}

Header^{۵۳}

Domain Name Requests^{۵۴}

سپس ارتباط آن با آدرس به دست آمده از درخواست بوده است. سپس ترافیک فرستاده شده از سمت کاربر برای آدرس هایی که به دست آمده اند بازبینی شده و جمع آوری می شود و در ادامه با استفاده از روشی مبتنی بر تطابق رشته های^{۵۵} درخواست، شباهت سنجی صورت گرفته و برنامه های بازبسته بندی شده مشخص می شوند. یکی از ایرادات این پژوهش آن است که مبتنی بر رفتار بدافزارهای بیشتر شناخته شده انجام شده است و در صورتی که بدافزاری رفتار مشابه با بدافزارهای محبوب نداشته باشد شناسایی نمی شود. همچنین تحلیل ترافیک شبکه ی کاربر در ترافیک خام و رمزگذاری نشده صورت می گیرد و در صورتی که ارتباطات بدافزار بازبسته بندی شده حاوی ترافیک رمزگذاری شده باشد، آنگاه روش پیشنهادی در تشخیص آن ها ناتوان خواهد بود.

استفاده از روش های ترکیب با تحلیل ترافیک شبکه نظیر بررسی دسترسی های کاربران، در پژوهش شارما و همکاران [۶۰] استفاده شده است. تشخیص برنامه های بازبسته بندی شده در این پژوهش مبتنی بر طبقه بندی برنامه ها با استفاده از تحلیل ترافیک شبکه و بررسی مجوزهای دسترسی مورد نیاز برنامه می باشد. پس از دیس اسمبل برنامه، مجوزهای دسترسی از فایل فراداده ی^{۵۶} برنامه مورد نظر استخراج شده و تشکیل یک بردار ویژگی دودویی را می دهند. این پژوهش از دو سطح برنامه های مورد نظر را بررسی می کند، در سطح اول اگر بردار ویژگی برنامه با برداری از برنامه های بازبسته بندی شده تطابق داشته باشد، آنگاه برنامه مورد نظر به عنوان یک برنامه مشکوک به سطح بعد فرستاده می شود. در سطح دوم، ترافیک رمزگذاری نشده ی برنامه تفکیک شده و تحلیل روی ترافیک TCP ادامه می یابد، سپس با استفاده از یک طبقه بند درخت تصمیم^{۵۷}، برنامه های بازبسته بندی شده با استفاده از تحلیل ترافیک کاربران، مشخص می شوند.

اگر چه پژوهش های صورت گرفته در این دسته معمولاً سرعت خوبی از نظر مقایسه ی مدل استخراج شده دارند، اما باید توجه داشت که محدودیت های موجود در این قسمت مانند به خطر افتادن حریم خصوصی کاربران، از طریق بررسی ترافیک رمزنگاری شده، یکی از ویژگی های منفی پژوهش های مبتنی بر ترافیک برنامه می باشد. از طرفی، بسیاری از برنامه های اندرویدی، به صورت برون خط فعالیت می کنند که موجب می شود روش های این دسته اساساً از بررسی آن ها ناکام بمانند.

روش های مبتنی بر تحلیل منابع

عمده ی روش های موجود در این دسته، فرض استفاده از منابع برنامه های بازبسته بندی شده را در تعریف بازبسته بندی اعمال کرده اند. در این دسته از تعریف بازبسته بندی، متقلب با تغییر کدهای برنامه و

String Matching^{۵۵}
Manifest File^{۵۶}
Decision Tree^{۵۷}

استفاده‌ی مستقیم از منابع آن، سعی در جعل برنامه اصلی دارد. در این حالت رابط کاربری برنامه مشابه با رابط کاربری برنامه اصلی است اما کارکرد منطقی آن دچار تغییرات زیادی می‌شود بنابراین شناسایی این دسته از برنامه‌ها با استفاده از ویژگی‌های مبتنی بر منابع آن انجام می‌شود. به عنوان اولین پژوهش از این دسته، شائو و همکاران [۱]، برای تشکیل امضا هر برنامه، از دو دسته ویژگی آماری^{۵۸} و ساختاری^{۵۹} استفاده می‌کنند. در قسمت ویژگی‌های آماری، ۱۵ ویژگی شاخص از منابع برنامه‌های اندرویدی نظیر تعداد فعالیت‌ها^{۶۰}، تعداد مجوزهای دسترسی، تعداد فیلترهای هدفمند^{۶۱} و میانگین تعداد فایل‌های png و xml استخراج می‌شود. علاوه بر این ۵ ویژگی، ۱۰ ویژگی دیگر آماری شامل میانگین فراخوانی به ۱۰ منبع پربازدید در برنامه‌های اندرویدی، مطابق با جدول ۳-۱، استخراج می‌شود.

جدول ۳-۱: ۱۰ منبع پربازدید در پژوهش [۱]

#	نوع منبع
1	id
2	drawable
3	string
4	color
5	style
6	dimen
7	layout
8	xml
9	integer
10	array

پس از استخراج ۱۵ ویژگی آماری، دو ویژگی ساختاری دیگر مبتنی بر لایه‌ی فعالیت و کنترل‌گر رویدادهای^{۶۲} برنامه ساخته می‌شود. برای ساخت این دو ویژگی از یک درخت انتزاعی مبتنی بر فعالیت‌های برنامه، استفاده شده‌است. در قسمت شباهت‌سنجی، پس از نرمال‌سازی ویژگی‌های استخراج شده در مراحل قبل، از دو روش طبقه‌بندی مبتنی بر نزدیک‌ترین همسایه و طبقه‌بندی طیفی^{۶۳} برای تشخیص برنامه‌های تقلبی استفاده شده‌است.

Statistical^{۵۸}
Structural^{۵۹}
Activities^{۶۰}
Intent Filters^{۶۱}
Event Handler^{۶۲}
Spectral Classification^{۶۳}

در پژوهش دیگری که توسط لین و همکاران [۶۱] انجام شده است، تشخیص بازبسته‌بندی با استفاده از چکیده‌ی تصاویر موجود در فایل‌های apk موجود در پوشه‌ی منابع، صورت می‌گیرد. در این پژوهش ابتدا چکیده‌ی تمامی تصاویر استخراج شده و شباهت‌سنجی میان آن‌ها به صورت تجمعی صورت می‌گیرد. در نهایت برنامه‌هایی که امتیاز تشابه آن‌ها از یک حد آستانه بیشتر باشد به عنوان برنامه‌های بازبسته‌بندی شده تشخیص داده می‌شوند. یکی از مفروضات پژوهش، به جهت کاهش فضای مقایسه، عدم لو رفتن کلید خصوصی توسعه‌دهنده‌ی برنامه می‌باشد. علاوه بر این، استفاده از روش‌های چکیده‌سازی معمول در این پژوهش، منجر به مقاومت پایین آن در مقابل مبهم‌نگاری می‌شود.

سان و همکاران [۶۲] با استفاده از تحلیل گراف فعالیت‌های برنامه، مجموعه‌ای از تصاویر ضبط‌شده^{۶۴} هر برنامه را تحت هر فعالیت استخراج کرده و هر دید^{۶۵} موجود در تصاویر را با استفاده از یک مستطیل نشان‌دهی و در نهایت به ازای هر تصویر، یک گروه از ناحیه دیدها ساخته می‌شود. در نهایت با استفاده از ادغام نواحی و گراف فعالیت‌های برنامه، گرافی تحت عنوان گراف گروه-ناحیه ساخته می‌شود. برای مقایسه و شباهت‌سنجی، با استفاده از پیمایش گراف‌ها، گره‌های گراف به عنوان نواحی برنامه با یکدیگر مقایسه می‌شود. دو ناحیه در صورتی با یکدیگر یکسان هستند که تعدادی از مجموعه‌های مدل‌شده به شکل مستطیل در این نواحی با یکدیگر هم‌پوشانی داشته باشند.

هو و همکاران [۶۳] با بررسی برنامه‌های بازبسته‌بندی شده متوجه شدند که ویژگی‌های ساختاری واسط‌های کاربری^{۶۶} همانند طول و اندازه‌ی اشکال و دکمه‌ها کم‌تر دچار تغییر می‌شود و بیشتر ویژگی‌های محتوایی مانند رنگ پس‌زمینه تغییر می‌کند. با توجه به این یافته، آن‌ها سه ویژگی مهم مبتنی بر ساختار واسط‌های کاربری را استخراج و تحت عنوان امضای برنامه استفاده می‌کنند و در ادامه، این سه ویژگی را به یک فعالیت نسبت می‌دهند. در نهایت با کنار هم قرار گرفتن سه‌تایی‌های هر فعالیت و مقایسه‌ی آن‌ها با یکدیگر برنامه‌های تقلبی مشخص می‌شوند.

هدف آقای لین در پژوهش [۶۴] کاهش نرخ منفی غلط و مثبت غلط بوده است. در این پژوهش از ساختار سلسله‌مراتبی^{۶۷} فایل‌های موجود در یک apk استفاده شده است. ساختار سلسله‌مراتبی مبتنی بر فایل‌های برنامه استخراج می‌شود و گره‌های این درخت برچسب‌گذاری می‌شود. برچسب‌گذاری درخت در این پژوهش، مبتنی بر نوع فایل مورد نظر آن گره انجام می‌شود چرا که برچسب‌گذاری بر اساس نام فایل‌ها، مقاومت پایینی در مقابل مبهم‌نگاری دارد و متقلب می‌تواند به راحتی روش را دور بزند. علاوه بر این مشکل دیگری که در این پژوهش وجود دارد، این است که در صورتی که چندین فایل از یک نوع وجود داشته باشد، گره‌های مشابه در درخت سلسله‌مراتبی به وجود می‌آید و در نهایت نرخ مثبت غلط به شدت افزایش پیدا

ScreenShot^{۶۴}

View^{۶۵}

User Interface^{۶۶}

File Structure^{۶۷}

می‌کند. برای حل این مشکل، نویسنده از یک ساختار چند مرحله‌ای برای تشخیص برنامه‌های بازبسته‌بندی شده استفاده می‌کند. در این ساختار ابتدا برچسب‌گذاری به نحوی انجام می‌شود که نرخ مثبت غلط زیادی در نتایج وجود داشته باشد و در ادامه از روشی با نرخ منفی غلط بالا استفاده می‌کند تا دقت پژوهش را افزایش دهد. شباهت‌سنجی درخت‌ها با استفاده از فاصله ویرایشی میان دو درخت انجام می‌شود. روش مورد نظر مقاومت پایینی در مقابل مبهم‌نگاری‌های ساده به خصوص افزونگی فایل‌ها، خواهد داشت چرا که متقلب می‌توان به راحتی با ایجاد تعداد زیادی فایل بلااستفاده، سیستم تشخیص را دور بزند.

گوآ و همکاران [۶۵] با استفاده از طرح‌بندی^{۶۸} و ایجاد درخت مبتنی بر آن برنامه‌های بازبسته‌بندی شده را تشخیص می‌دهند. ابتدا درختی از تمامی مولفه‌های از نوع منبع و با استفاده از فایل‌های *xml*، به نام درخت طرح‌بندی کامل^{۶۹}، استخراج می‌شود. درخت مذکور حاوی تمامی مولفه‌های طرح‌بندی یک برنامه اندرویدی است که تنها یک ریشه دارد. در ادامه برای مقایسه و شباهت‌سنجی درخت طرح‌بندی کامل، از یک روش مبتنی بر چکیده‌سازی *CTPH* استفاده شده‌است.

استفاده از طرح‌بندی در پژوهش‌های مبتنی بر منابع محبوبیت زیادی دارد. برای مثال، در روش لیو و همکاران [۶۶]، ابتدا طرح‌بندی‌های مربوط به کتابخانه‌های اندرویدی حذف شده، سپس هر طرح‌بندی به یک درخت نگاشت می‌شود. به جهت مقایسه، طرح‌بندی‌های برنامه با یکدیگر ادغام می‌شوند و درخت نهایی در قالب یک فایل *xml* کدگذاری شده و در ادامه با استفاده از مقایسه‌ی مبتنی بر چکیده‌سازی فازی فایل‌های *xml*، برنامه‌های بازبسته‌بندی شده مشخص می‌شوند. در نمونه‌ی دیگری، لیو و همکاران [۶۷] در یک ساختار دو مرحله‌ای مبتنی بر طرح‌بندی، برنامه‌های بازبسته‌بندی شده را تشخیص می‌دهند. مرحله‌ی اول در این پژوهش شامل تشخیص برنامه‌های مشکوک به صورت درشت‌دانه^{۷۰} است که با استفاده از تصاویر موجود در پوشه‌ی منابع انجام می‌گیرد. در ادامه و به صورت ریزدانه، درخت طرح‌بندی تشکیل شده و امضای هر برنامه را تشکیل می‌دهد. برای مقایسه درخت‌ها، از روش‌های مرسوم مانند فاصله ویرایشی، استفاده شده‌است.

به عنوان یکی از آخرین پژوهش‌های موجود در این دسته به جهت بهبود روش‌های [۶۸، ۶۲] ما و همکاران [۶۹] با معرفی گراف انتزاعی طرح‌بندی^{۷۱} و گراف طرح‌بندی انتقالی^{۷۲}، برنامه‌های بازبسته‌بندی شده را شناسایی می‌کنند. در این پژوهش در ابتدا طرح‌بندی‌های هر برنامه تحت عنوان یک گراف جهت‌دار مدل می‌شود که هر گره گراف مذکور شامل طرح‌بندی‌های مشابه می‌باشد. در مقایسه با [۶۸]، روش ما و همکاران، از سرعت بیشتری در کدگذاری طرح‌بندی و تبدیل آن‌ها به گراف انتزاعی برخوردار است. علاوه

^{۶۸} Layout
^{۶۹} Total tree layout
^{۷۰} Coars Grain
^{۷۱} Abstract Layout Graph
^{۷۲} Abstract Transition Graph

بر این در قسمت شباهت سنجی و مقایسه، از روشی بهبودیافته، مبتنی بر یافتن تطبیق گراف بیشینه^{۷۳} استفاده شده که ارزیابی عملکرد پژوهش نسبت به دو پژوهش مورد مقایسه، افزایش سرعت و دقت را به همراه داشته است.

۳-۳-۲ مبتنی بر تحلیل پویا

به صورت کلی پژوهش‌های صورت گرفته در این قسمت را می‌توان به دو بخش روش‌های مبتنی بر جعبه‌شن^{۷۴} و یا روش‌های خودکار تقسیم‌بندی کرد. در روش‌های مبتنی بر جعبه‌شن، پژوهش‌کنندگان با ایجاد محیطی شبیه‌سازی شده و تعامل حداکثری با برنامه‌اندویدی مدنظر، سعی در شبیه‌سازی رفتار برنامه و جمع‌آوری مجموعه‌ای از ویژگی‌های موردنیاز به جهت مقایسه با یکدیگر دارند. این دسته از روش‌های پویا دو ایراد اساسی دارند، نخست آن‌که ایجاد یک محیط شبیه‌سازی شده و اجرای هر برنامه به طوری که تمامی رفتارهای آن‌را شبیه‌سازی کند، نیازمند زمان زیادی است که ممکن است روش را ناکارآمد کرده و بررسی برنامه‌های موجود را محدود نماید. علاوه بر این ایجاد محیطی که بتواند به صورتی کامل رفتار برنامه را شبیه‌سازی کند نیازمند آن‌است که به نوعی تمامی خدمات هر برنامه منحصرا بررسی شود که این عملی تقریباً ناممکن است مگر آن‌که شبیه‌سازی منحصر یک برنامه خاص پیاده‌سازی شود. ایراد دوم آن‌است که برنامه‌های بازبسته‌بندی شده که دارای بدافزار هستند، ممکن است پیاده‌سازی آن‌ها قادر به شناسایی محیط شبیه‌سازی شده باشد و در نتیجه، از اجرای قسمت‌هایی از برنامه خودداری کرده و سیستم تشخیص را به خطا بیندازد. برخلاف روش‌های مبتنی بر جعبه‌شن، روش‌های مبتنی بر تحلیل خودکار بیشتر سعی در استفاده از داده‌های کاربران واقعی برنامه‌های اندرویدی دارند. بیشتر روش‌ها در این دسته از معماری‌های ابری و کارخواه-کارگزار استفاده کرده و با استفاده از ارسال گزارشات^{۷۵} از سمت کاربران شباهت سنجی را انجام می‌دهند. چندین نمونه از این روش‌ها را مانند [۳، ۴۲] در قسمت‌های پیشین بررسی کردیم. روش‌های مبتنی بر تحلیل خودکار هیچ شبیه‌سازی در مورد برنامه انجام نمی‌دهند و رفتار برنامه را با توجه به رفتار کاربران مورد تحلیل قرار می‌دهند. تمرکز ما در این قسمت روی پژوهش‌هایی است که بیشتر تمرکز آن‌ها بر پیاده‌سازی روشی مبتنی بر جعبه‌شن و شبیه‌سازی رفتار برنامه‌ها بوده‌است.

در پژوهش والریو و همکاران [۷۰] با استفاده از اجرای برنامه‌های اندرویدی در چندین جعبه‌شن مشهور و جمع‌آوری اطلاعات برای هر برنامه اندرویدی، یک پروفایل-استفاده^{۷۶} ساخته می‌شود. این پروفایل نشان‌دهنده‌ی استفاده‌ی برنامه از ۸ خدمت مبتنی بر منابع سیستمی شامل لیست تماس‌ها، موقعیت

^{۷۳} Maximum Graph Matching

^{۷۴} SandBoxing

^{۷۵} Logs

^{۷۶} Usage Profile

مکانی، پیامک^{۷۷}، شبکه‌ی ارتباطی بی‌سیم^{۷۸}، برنامه‌ها، باتری و تماس‌ها می‌باشد. روند کلی جمع‌آوری امضای برنامه به این شکل است که برنامه‌های موجود در مخزن توسط یک مولفه در چندین جعبه‌شن مشهور و در دسترس اجرا شده و ویژگی‌های ذکر شده استخراج می‌شود و در نهایت تشکیل یک بردار ویژگی می‌دهند. در مرحله‌ی انتهایی، بردارهای ویژگی برنامه‌های اندرویدی با یکدیگر مقایسه شده و بسته‌های بازبسته‌بندی شده مشخص خواهند شد.

از آن‌جا که بررسی پردازش برنامه‌های اندرویدی به ازای ورودی‌های یکسان می‌تواند نتایج مشابه در برنامه‌های اندرویدی داشته باشد، ژانگ و همکاران^[۷۱] روشی را مبتنی بر زمان مورد نیاز برای اجرای ورودی‌های یکسان توسط پردازنده^{۷۹}، پیاده‌سازی کرده‌اند. در این پژوهش، ابتدا تعدادی ورودی به هر برنامه اندرویدی موجود در مخزن برای اجرا داده می‌شود و پس از اتمام پردازش، بازه‌ی زمانی اجرای پردازش در پردازنده ضبط می‌شود و تحت عنوان یک دوتایی <پردازش، زمان> نمایش داده می‌شود. ورودی‌های یکسان برای تمامی برنامه‌های مورد بررسی اجرا می‌شود و در نهایت لیستی از دوتایی‌های موجود برای هر برنامه، تشکیل امضای آن را می‌دهد.

در روش دیگری که توسط نگویان و همکاران^[۷۲] اجرا شده، برنامه‌های اندرویدی با یک روش مبتنی بر جعبه‌شن و استخراج ویژگی‌هایی از واسط‌کاری برنامه، بسته‌های بازبسته‌بندی شده را شناسایی می‌کنند. در این پژوهش با استفاده از یک جعبه‌شن که وظیفه‌ی آن ضبط تصاویر محیط کاربر برنامه می‌باشد، تصاویری از برنامه‌های اندرویدی مورد بررسی ذخیره می‌شود. از آنجایی که روش‌های چکیده‌سازی مرسوم، چکیده‌ای کاملاً متفاوت حتی برای کوچکترین تغییرات تولید می‌کنند، در این پژوهش از روشی مبتنی بر چکیده‌سازی ادراکی^{۸۰} استفاده شده‌است. در این دسته از روش‌های چکیده‌سازی، تغییرات کوچک به همان اندازه باعث تغییرات کوچک در چکیده‌ی تولیدشده می‌شود و فایل‌های مشابه اما با تغییرات اندک می‌توانند پس از مقایسه‌ی چکیده‌سازی ادراکی، شناسایی شوند. در مرحله‌ی شباهت‌سنجی، تصاویر ضبط‌شده از هر برنامه، با استفاده از روش‌های میانگین چکیده‌سازی^{۸۱}، با تصاویر برنامه‌های موجود در مخزن مورد مقایسه قرار می‌گیرند و در نهایت برنامه‌های بازبسته‌بندی شده به این طریق تشخیص داده می‌شوند.

در پژوهش دیگری که مبتنی بر واسط‌های کاربری انجام شده‌است، سو و همکاران^[۷۳] با استفاده از استخراج فعالیت‌های برنامه موجود در فایل فراداده‌ی آن‌ها، فعالیت‌های مورد نظر را در یک محیط مبتنی بر جعبه‌شن اجرا و تصاویر حاصل از اجرای هر فعالیت را ضبط می‌کنند. در قسمت شباهت‌سنجی،

SMS^{۷۷}

Wifi^{۷۸}

CPU^{۷۹}

Perception Hashing-Phash^{۸۰}

Average Hashing^{۸۱}

اطلاعاتی از درون هر کدام از تصاویر ضبط شده از برنامه‌ها در حین اجرای آن، استخراج می‌شود و تشکیل بردار ویژگی هر برنامه را می‌دهند. در ادامه با استفاده از روش‌های مبتنی بر چکیده‌سازی محلی^{۸۲}، شباهت بردارهای ویژگی با یکدیگر سنجیده و در صورتی که میزان شباهت از یک حد آستانه بیشتر باشد، جفت مورد بررسی، بازبسته‌بندی شده یکدیگر شناسایی می‌شوند. برنامه‌ها برای مقاومت بیشتر در مقابل مبهم‌نگاری از لیست فعالیت‌های برنامه استفاده کرده و نقاط ورود^{۸۳} را از روی این لیست فعالیت مشخص می‌کند. این پژوهش، فعالیت‌های موجود در برنامه‌های اندرویدی را مستقل از یکدیگر در نظر گرفته‌است و به همین دلیل نقاط ورود متنوعی مبتنی بر هر فعالیت تعیین می‌شود، در صورتی که در اکثر برنامه‌های اندرویدی، فعالیت‌های موجود کاملاً به هم وابسته هستند و گاهی یک فعالیت نتیجه‌ی یک فعالیت دیگر است بنابراین فرض پژوهش در این قسمت به نظر اشتباه می‌باشد.

به طور کلی، تشخیص برنامه‌های اندرویدی مبتنی بر روش‌های پویا شامل استخراج ویژگی‌های متنوع ساختاری و رفتاری برنامه در حین اجرای آن در محیط شبیه‌سازی شده است. برای مثال، در پژوهش بلاسینگ و همکاران [۷۴] با استفاده از یک برنامه تزریق شده به هسته^{۸۴}ی سیستم عامل، تمامی فراخوانی‌های سیستمی ناشی از اجرای برنامه در محیط جعبه‌شن، استخراج می‌شود. استخراج فراخوانی سیستمی از طریق هسته‌ی سیستم عامل، منجر به مقاومت بالای این روش مقابل مبهم‌نگاری می‌شود. کیم و همکاران [۷۵]، برای تشخیص برنامه‌های بازبسته‌بندی شده، از بردارهای ویژگی مبتنی بر فراخوانی‌های واسط‌های برنامه‌نویسی^{۸۵} استفاده کرده‌اند و در نهایت با مقایسه‌ی برنامه‌ها، مسئله را حل می‌کنند. گوان و همکاران [۷۶]، با استفاده از یک روش مبتنی بر معناشناسی^{۸۶} در برنامه‌های اندرویدی، بسته‌های بازبسته‌بندی شده را شناسایی می‌کنند. آن‌ها با بررسی مشکلات تشخیص در تحلیل‌های ایستا، در ابتدا گراف جریان ارتباطی مبانی کلاس‌های برنامه را استخراج کرده و در ادامه با استفاده از وزن‌دهی گراف، مبتنی بر ورودی و خروجی کلاس، کلاس‌های اصلی برنامه را از کلاس‌های فرعی جدا می‌کنند. سپس متدهای هر کلاس با استفاده از گراف جریان میان متدهای کلاسی، به صورت نمادین اجرا می‌شوند و با بررسی تمامی جایگشت‌های ممکن به عنوان ورودی هر متد و خروجی آن‌ها، متدهای یکسان و شبیه به هم به دست می‌آیند. تشخیص بازبسته‌بندی با استفاده از یک حد آستانه، بر اساس تعداد متدهای یکسان مشخص می‌شود. یو و همکاران [۶۸] با استفاده از ساخت گراف فعالیت مبتنی بر واسط کاربری هر برنامه و در نهایت مقایسه‌ی گرافی، برنامه‌های بازبسته‌بندی شده را تشخیص داده‌اند. برای مقایسه‌ی گرافی، پس از وزن‌دهی گراف بر اساس تعداد تکرار هر فعالیت، از یک الگوریتم تطبیق بیشینه‌ی گراف، استفاده شده‌است.

Locality Sensitive Hash^{۸۲}

Entry Point^{۸۳}

Kernel^{۸۴}

API Calls^{۸۵}

Semantic^{۸۶}

۳-۳-۳ سایر روش‌ها

وانگ و همکاران [۷۷] از روشی مبتنی بر طبقه‌بندی برای شناسایی کتابخانه‌های اندرویدی استفاده کرده‌اند. با توجه به این که کتابخانه‌های اندرویدی در برنامه‌های مختلف تکرار می‌شوند بنابراین جمع‌آوری دنباله‌ی فراخوانی واسطه‌های برنامه‌نویسی در کتابخانه‌های مشابه، منجر به ساخته‌شدن دنباله‌های مشابه می‌شود. بنابراین پس از جمع‌آوری دنباله‌ی فراخوانی‌ها از برنامه‌های متعدد، طبقه‌بندی روی دنباله‌ی ورودی با استفاده از داده‌ی آموزشی انجام می‌شود و در نهایت کدهای کتابخانه‌ای با استفاده از این روش جدا می‌شوند. در قسمت دوم از یک راه حل دو مرحله‌ای برای تشخیص برنامه‌های اندرویدی بازبسته‌بندی شده استفاده شده‌است. پس از حذف کدهای کتابخانه‌ای، دنباله‌های فراخوانی واسطه‌های اندرویدی در هر برنامه محاسبه و دنباله‌های موجود با یکدیگر مقایسه می‌شوند. در این مرحله، برنامه‌های مشکوک به بازبسته‌بندی تشخیص داده می‌شوند و در مرحله‌ی بعدی بررسی دقیق‌تر صورت می‌گیرد. در مرحله‌ی دوم، متغیرهای موجود در برنامه در روشی مبتنی بر توکن^{۸۷}، به صورت برداری نشان داده می‌شوند. به جهت شباهت سنجی و مقایسه‌ی بردارهای موجود، تعداد بردارهای یکسان، که مبتنی بر شناسه‌ی هر متغیر و نام آن است، با یکدیگر مقایسه می‌شود و در صورتی که تعداد توکن‌های مشابه، در یک جفت برنامه از یک حد آستانه بیشتر باشد، جفت مورد بررسی به عنوان بازبسته‌بندی در نظر گرفته می‌شوند. روش پژوهش، از آن‌جا که در دو مرحله برنامه‌های بازبسته‌بندی شده را شناسایی می‌کند از سرعت خوبی برخوردار است، اما با استفاده از روش‌های مبتنی بر افزودن متغیرهای بیهوده، می‌توان روش یادشده را دور زد.

نیشا و همکاران [۷۸]، با استفاده از ویژگی‌های مبتنی بر مجوزهای دسترسی و ایجاد بردار دودویی مجوزهای هر برنامه، به وسیله‌ی روش‌های یادگیری ماشین برنامه‌های بازبسته‌بندی شده را تشخیص می‌دهند. یکی از مشکلات روش موجود آن است که برای ایجاد مجموعه‌ی داده آزمون، از یک روش مبتنی بر کپی مجدد برنامه‌های اصلی استفاده شده‌است که باعث می‌شود روش موجود در عین داشتن دقت بالا در مورد مجموعه‌ی آزمون پژوهش، به صورت کلی مقاومت بالایی در تشخیص بازبسته‌بندی نداشته باشد. مشکل دیگر این روش، آن است که عملاً تعریف تشخیص بازبسته‌بندی در این روش‌ها، حل مسئله‌ی تصمیم در مورد بازبسته‌بندی است و شامل تشخیص جفت تقلبی نمی‌شود.

۳-۴ پیش‌گیری از بازبسته‌بندی

دیدگاه پژوهش‌های پیشین در زمینه‌ی تشخیص برنامه‌های بازبسته‌بندی شده، بررسی برنامه‌های مشکوک پس از بازبسته‌بندی توسط مهاجم بوده‌است اما عمده‌ی پژوهش‌های موجود به جهت پیش‌گیری از بازبسته‌بندی

^{۸۷}Token

نیازمند ایجاد تغییراتی در برنامهک پیش از انتشار آن به صورت عمومی می‌باشد. این روش‌ها، از بازبسته‌بندی برنامهک توسط متقلبان جلوگیری کرده و یا در صورتی که بازبسته‌بندی صورت بگیرد توسعه‌دهندگان متوجه آن خواهند شد. مزیت بزرگ روش‌های پیش‌گیری از بازبسته‌بندی آن است که از قطعیت بیشتری برخوردار هستند، یعنی در صورتی که بازبسته‌بندی صورت بگیرد اکثراً به صورت قطعی توانایی تشخیص و جلوگیری از انتشار برنامهک‌ها را دارند. به صورت کلی پژوهش‌های اخیر در این حوزه را می‌توان به دو دسته تقسیم کرد که در ادامه به توضیح مختصری از هر کدام و شرح پژوهش‌های هر دسته می‌پردازیم.

۳-۴-۱ روش‌های مبتنی بر نشان‌گذاری

در این دسته از روش‌ها که نیازمند ایجاد تغییراتی در کد برنامهک توسعه‌داده‌شده است، توسعه‌دهندگان قسمتی از امضای خود را تحت عنوان حق‌تکثیر^{۸۸} که با نام نشان شناسایی می‌شود، در برنامهک پنهان می‌سازند و در صورتی که برنامهک جدیدی بازبسته‌بندی شود با توجه به تغییر این حق‌تکثیر قادر به تشخیص و جلوگیری از بازبسته‌بندی خواهند بود. به صورت کلی روش‌های نشان‌گذاری^{۸۹} به دو نوع ایستا و پویا تقسیم می‌شوند. در روش‌های ایستا، پیش از ساخته‌شدن برنامهک و انتشار آن، حق‌تکثیر ایجاد و در ساختار داده‌ای و یا کدهای برنامهک تزریق می‌شود اما در روش‌های پویا، به جهت غلبه بر محدودیت روش‌های ایستا و بررسی دقیق‌تر، حق‌تکثیر توسعه‌دهندگان در حین اجرای برنامهک توسط کدهای مورد توسعه ایجاد می‌شود و در ساختار آن قرار می‌گیرد. در روش‌های مبتنی بر نشان‌گذاری از یک روند مشخص برای شناسایی و تایید نشان تزریق‌شده به برنامهک‌ها استفاده می‌شود که عدم بازبسته‌بندی را پیش از اجرا و یا در حین اجرا، بسته به ایستا و یا پویا بودن روش، مشخص می‌کند. از طرفی به دلیل عدم قطعیت تشخیص در عمده‌ی روش‌های تشخیص، روش‌های نشان‌گذاری و روش‌های تشخیص برنامهک‌های بازبسته‌بندی شده می‌توانند به صورت مکمل یکدیگر عمل کنند. در این حالت ابتدا برنامهک‌های مشکوک با استفاده از روش‌های بیان‌شده در حوزه‌ی تشخیص، شناسایی می‌شوند و در ادامه به جهت اطمینان از صحت برنامهک مورد نظر، توسط روش‌های مبتنی بر نشان‌گذاری، حق‌تکثیر توسعه‌دهندگان بررسی می‌شود و در صورتی که برنامهک مدنظر بازبسته‌بندی شده باشد، از فروشگاه حذف خواهد شد. استفاده از هر دوی این روش‌های سرعت و دقت تشخیص برنامهک‌های اندرویدی بازبسته‌بندی شده را افزایش می‌دهد.

یکی از اولین پژوهش‌های این حوزه توسط وو و همکاران [۷۹] انجام شده‌است. در این پژوهش ابتدا رشته‌ای از اعداد و حروف برای تولید نشان مشخص می‌شود. سپس رقم مشخص شده در فرایندی مشخص به یک گراف جایگشت نگاشت شده و در ادامه گراف مورد نظر با استفاده از یک قطعه‌کد نشان‌داده

می‌شود. برای جلوگیری از مبهم‌نگاری نشان در فرایند بازبسته‌بندی، کد ایجادشده برای گراف نشان‌گذاری، در قسمت‌های مختلف برنامه‌کد هدف تقسیم می‌شود. برای تایید برنامه‌کد اصلی و جلوگیری از بازبسته‌بندی از یک برنامه‌کد ثانویه که حکم فراداده‌ی برنامه‌کد اصلی را دارد استفاده می‌شود. برای ساخت برنامه‌کد ثانویه، از نمونه‌ی آزمونی که تمامی کد را پوشش دهد استفاده می‌شود هر ورودی آزمون در این نمونه، یک گرداننده دارد که هر تکه از کد نشان‌گذاری شده در قسمت قبل درون آن قرار می‌گیرد. در واقع هدف از تولید برنامه‌کد ثانویه، نگه‌داشت مکان‌هایی از برنامه‌کد اصلی است که کد نشان‌گذاری در آن قسمت‌ها قرار گرفته‌است چون همانطور که اشاره کردیم جهت جلوگیری از مبهم‌نگاری نشان‌گذاری در حین بازبسته‌بندی، کدهای نشان‌گذاری در قسمت‌های مختلف برنامه‌کد اصلی تقسیم می‌شوند. بنابراین در صورتی که برنامه‌کد ثانویه، در دسترس متقلب قرار گیرد، می‌تواند تمامی نشان‌گذاری‌های موجود در کد را شناسایی و آن‌ها را حذف کند بنابراین توسعه‌دهندگان نمی‌توانند از بازبسته‌بندی جلوگیری کنند.

ژانگ و همکاران [۸۰]، از یک روش نشان‌گذاری مبتنی بر تصاویر نشان‌گذاری شده استفاده می‌کنند. نویسندگان، با توجه به این که در پژوهش [۷۹] هیچ وابستگی داده‌ای میان کدهای نشان‌گذاری شده وجود ندارد، روشی را پیشنهاد داده‌اند تا مهاجم نتواند با تحلیل برنامه‌کد و یافتن کدهای نشان‌گذاری شده، در آن‌ها مبهم‌نگاری ایجاد کرده و سیستم تشخیص را دور بزند. در این روش، با استفاده از کاراکترهای اسکی^{۹۰}، تصویر ورودی را کدگذاری کرده و طی یک دنباله‌ی تصادفی از رخدادها، کد تولیدشده‌ی اسکی در مسیر رخدادهای تولیدشده تقسیم می‌شود. به جهت این‌که از تغییر نشان‌ها توسط مهاجم جلوگیری شود، تعدادی از کاراکترهای تصویر ورودی را با اعداد ثابت موجود در کد برنامه‌کد جایگزین می‌سازند که موجب می‌شود تصویر ورودی تغییر چندانی نکند و در عین حال وابستگی داده‌ای میان کدهای برنامه‌کد و تصویر ورودی ایجاد شود. یکی از مشکلات دو روش اخیر، عدم امکان بررسی اصالت برنامه‌کد توسط کاربران می‌باشد. برای رفع این مشکل رن و همکاران [۸۱] از کدگذاری بلوکی مبتنی بر ساختارهای خودرمزگشایی^{۹۱} استفاده کرده‌اند. در این پژوهش، کد نشان‌گذاری شده پس از تولید، در بلوک‌های شرطی کد برنامه‌کد مورد توسعه قرار گرفته و رمزگذاری صورت می‌گیرد و در حین اجرا رمزگشایی بلوک‌های اجرایی منجر به تایید اصالت کد نشان‌گذاری شده می‌شود.

یکی از مشکلات روش‌های مطرح‌شده در این زمینه آن‌است که جلوگیری از بازبسته‌بندی همواره باید توسط توسعه‌دهندگان صورت گیرد. به همین جهت ایجاد روندی خودکار جهت جلوگیری از اجرای برنامه‌کدهای بازبسته‌بندی شده توسط خود برنامه‌کد ایده‌ای بود که در پژوهش [۸۲] توسط سان و همکاران دنبال شده‌است. در این پژوهش ابتدا تمامی ساختارهای شرطی^{۹۲} برنامه‌کد استخراج می‌شود و با استفاده

Ascii^{۹۰}

Self Decryption^{۹۱}

Conditional Statement^{۹۲}

از تحلیل و تغییر شروط به صورتی که منطق برنامه تغییر نکند، کد نشان‌گذاری شده در ساختار برنامه جایگذاری می‌شود. در نهایت با استفاده از یک جریان مبتنی بر بارگذاری پویا، در صورتی که کد نشان‌گذاری شده تغییری کرده‌باشد، از اجرای برنامه جلوگیری خواهد شد.

لوآ و همکاران [۸۳] در سال ۲۰۱۶ روشی را مبتنی بر کلید عمومی توسعه‌دهنده‌ی برنامه اندرویدی، برای جلوگیری از بازبسته‌بندی ارائه کرده‌اند. نویسندگان در طی روشی به نام شبکه‌ی پنهانی مرتبط^{۹۳}، به عنوان ورودی، کد برنامه اندرویدی به همراه کلید عمومی کاربر توسعه‌دهنده‌ی برنامه را دریافت می‌کند و با استفاده مقایسه‌ی کلید عمومی ثبت‌شده در داخل برنامه و زیررشته‌ای از محاسبات مبهم‌نگاری شده در برنامه هنگام اجرای آن، اصالت برنامه را تشخیص می‌دهند. برای به دست آوردن کلید عمومی در حین اجرای برنامه، از ویژگی بازتاب در زبان جاوا استفاده شده‌است. یکی از مشکلات این پژوهش آن‌است که به دلیل استفاده از ویژگی بازتاب برای دستیابی به کلید عمومی، حمله‌کننده می‌تواند قسمتی که متد فراخوانی می‌شود را بازنویسی کرده و از طریق ورودی‌های قبلی، متد را بازفراخوانی کند. در این صورت حمله‌کننده به راحتی می‌تواند به کلید عمومی دسترسی پیدا کرده و در ادامه با کمی تحلیل ایستا، قسمتی از برنامه را که کلید عمومی در آن ثابت شده‌است را بیابد. به جهت مقابله با این نوع از حملات، زنگ [۸۴] راه‌حلی مبتنی بر بمب‌های منطقی^{۹۴} ارائه کرده‌است. بمب‌های منطقی، قطعه کدهای کوچکی هستند که تنها در حالاتی خاص فعال می‌شوند. در طی این روش، کدهای نشان‌گذاری در ساختار شرط‌های منطقی در برنامه تزریق می‌شود به صورتی که برخی از این شروط در طی اجرای برنامه در سیستم عامل کاربران معمولی اجرا می‌شود. در صورتی که مهاجم از روش‌های مبتنی بر جعبه‌سیاه استفاده ننماید، در آن صورت محیط اجرای وی محدود به تعدادی سیستم عامل و ویژگی‌های مبتنی بر آن خواهد بود، این در حالی است که به صورت کلی کاربران گستردگی بیشتری در تنوع این ویژگی‌ها دارند. بنابراین می‌توان شروط منطقی این بمب‌ها را به گونه‌ای نوشت، که کاربر مهاجم شانس کمی برای دسترسی به نشان‌گذاری داشته‌باشد. با استفاده از این روش، پژوهش [۸۳] توسعه یافت و تا حدودی مشکلات آن برطرف گردید، اما پژوهش جدید نیز دچار مشکلاتی شد که در ادامه به آن خواهیم پرداخت. در صورتی که کاربر بتواند با تحلیل پویا و فعال‌سازی شروط و در نهایت دسترسی به بلوک بمب‌های منطقی، پیش از اجرای کدهای رمزگشایی شده آن‌ها را تغییر دهد، آن‌گاه می‌تواند از تشخیص اصالت برنامه جلوگیری نماید. تانر و همکاران [۸۵] برای حل این مشکل از کدهای بومی استفاده کرده‌اند. از آنجایی که تغییر کدهای بومی در حین بازبسته‌بندی برای مهاجم دشوار و پرهزینه‌است، نویسندگان پس از دستیابی به کدهای بومی و کدهای اصلی برنامه، مجموعه‌ای از بایت‌کدهای دالویک رمزنگاری شده را در قسمت‌های مختلف برنامه اندرویدی اصلی تزریق کرده و متدهای بررسی صحت برنامه را در کدهای بومی پیاده‌سازی می‌کنند. رمزنگاری بایت‌کدها در کنار

Stochastic Stealthy Network^{۹۳}

Logic Bomb^{۹۴}

فراخوانی توابع بررسی صحت در کدهای بومی، موجب شده است که روش موجود، در مقابل تحلیل ایستا و پویا توسط مهاجم امنیت داشته باشد.

۳-۴-۲ روش‌های مبتنی بر مبهم‌نگاری

همانطور که در فصل ۲ اشاره شد، مبهم‌نگاری یکی از روش‌های پیش‌رو برای جلوگیری از بازبسته‌بندی توسط مهاجمان است. توسعه‌دهندگان از مبهم‌نگاری استفاده می‌کنند تا تغییر دادن برنامه‌های اندرویدی خود را سخت و هزینه‌بر کنند. به صورت کلی مبهم‌نگاری در وهله‌ی اول موجب می‌شود تا ساختار منطقی اصلی و به طور مشخص‌تر، بدنه‌ای اصلی منطقی برنامه از دید مهاجمان مخفی بماند. در مرحله‌ی بعدی در صورتی که مهاجم بتواند بدنه‌ی منطقی اصلی برنامه اندرویدی قربانی را شناسایی کند، تغییر دادن آن و سپس پیاده‌سازی منطق مهاجم به صورت یک بدافزار، کاری سخت، زمان‌بر و هزینه‌بر خواهد بود، به طوری که ممکن است مهاجم از انجام این عمل امتناع ورزد. همانطور که در فصل پیش دیدیم، برخی از ابزارهای موجود، مبهم‌نگاری در برنامه‌های اندرویدی را انجام می‌دهند. اما پژوهش‌های موجود در این قسمت نیز، به صورت پیش‌رو، روش‌هایی را توسعه داده‌اند که می‌توانند نقش پیش‌گیری در بازبسته‌بندی برنامه‌های اندرویدی را داشته باشند.

یکی از روش‌های مبهم‌نگاری در برنامه‌های اندرویدی، تغییر نام شناسه‌ها، کلاس‌ها و متدهای برنامه می‌باشد. این روش موجب می‌شود تا تحلیل و جست‌وجوی رشته‌ها در برنامه دشوار شده و مهاجم نتواند با تحلیل برنامه، آن را تغییر دهد. از آنجایی که تغییر نام در بیشتر مواقع تاثیری بر منطق برنامه ندارد، تغییر نام شناسه‌های برنامه، روشی است که ورمک و همکاران [۸۶] از آن استفاده کرده‌اند. پروگارد ابزار دیگری است که با تغییر نام کلاس‌ها، متدها و شناسه‌های موجود در برنامه مبهم‌نگاری را انجام می‌دهد. تغییر نام در پروگارد با استفاده از کاراکترهای محدود اسکی صورت می‌گیرد، به همین جهت دکس‌گارد معرفی شد تا نسخه‌ی پیشرفته‌تری از پروگارد را ارائه دهد که تغییر نام در آن‌ها با دامنه‌ی بیشتری از کاراکترهای اسکی انجام شود [۸۷].

برنامه‌های مبتنی بر زبان جاوا، با استفاده از پیشنهادهای مشخص^{۹۵} گروه‌بندی می‌شوند و در یک مسیر جاری قرار می‌گیرند که موجب می‌شود کلاس‌های موجود در یک گروه به صورت مستقیم به کلاس‌های هم‌گروهی خود دسترسی داشته باشند. تغییر نام بسته‌های اندرویدی، یکی از روش‌های معمول به جهت جلوگیری از بازبسته‌بندی است چرا که موجب می‌شود تحلیل برنامه برای مهاجم سخت شود. وانگ و همکاران [۸۸] از این روش برای افزایش مقاومت مقابل مبهم‌نگاری استفاده کرده‌اند. در روشی که توسط نویسندگان پیاده‌سازی شده است، تعدادی کلاس کمکی برای مخفی کردن کلاس‌های اصلی برنامه به

^{۹۵} NameSpace

بسته‌های نرم‌افزاری اضافه شده‌است. پژوهش [۸۷] نیز در تولید ابزار پروگارد از تغییر نام بسته‌ها استفاده کرده است.

حذف کدهای بومی‌ای که در هنگام اجرای برنامه نیازمند آن‌ها نیستیم، روش دیگری از مبهم‌نگاری برنامه‌های اندرویدی است که در پژوهش علم و همکاران [۸۹] از آن استفاده شده‌است. حذف کدهای بومی موجب می‌شود که خوانایی برنامه در حین دی‌کامپایل توسط مهاجمان کاهش یابد. علاوه بر این موضوع، به جهت خوانایی پایین کدهای بومی نسبت به بایت‌کدهای دالویک که یک زبان میانی محسوب می‌شود، در پژوهش علم و همکاران [۸۹]، نویسندگان قسمتی از برنامه را که شامل الگوریتم‌های رمزنگاری می‌شود، به صورت کدهای بومی پیاده‌سازی کرده‌اند که موجب می‌شود خوانایی این الگوریتم‌ها و در نتیجه دسترسی به الگوریتم‌های رمزگشایی سخت شود.

تغییر گراف کنترل جریان، روشی دیگری است که برای مبهم‌نگاری از آن استفاده می‌شود. تحلیل گراف کنترل جریان توسط مهاجمان، یکی از روش‌های محبوب بدافزارنویسان برای وارد ساختن بدافزار و بازبسته‌بندی است. گراف کنترل جریان نشان‌دهنده‌ی روال فراخوانی میان متدهای یک برنامه را نشان می‌دهد. باسی و همکاران [۹۰] با استفاده از افزودن یال به گراف جریان، آن را تغییر می‌دهند. افزودن یال با استفاده از اضافه کردن یک گره جدید در میان هر فراخوانی و ایجاد فراخوانی واسط انجام می‌شود. پردا و همکاران [۹۱] از همین روش برای تغییر گراف جریان استفاده کرده‌اند با این تفاوت که متد میانی اضافه‌شده، به صورت ایستا و با همان ورودی‌های متد اصلی تعریف می‌شود تا شناسایی متدهای اصلی فراخواننده دشوار شود. افزودن کدهای بی‌هوده، روشی دیگری است که در پژوهش‌های [۹۲، ۹۳، ۹۴]، استفاده شده‌است. در پژوهش لی و همکاران [۹۲]، با استفاده از افزودن کدهای nop پرش‌های قطعی^{۹۶} و ثبات‌های داده^{۹۷} پردازش‌های بی‌هوده‌ای در برنامه اضافه می‌کنند که منطق برنامه را تغییر نمی‌دهد. راستوگی [۹۴]، حالتی از دستورات شرطی را معرفی کرد که همواره یک نتیجه می‌دهد اما گراف جریان را به دو شاخه تقسیم می‌کند. در این حالت یکی از حلقه‌های این دستور، شامل کد اصلی خواهد بود و حلقه‌ی دیگر شامل پرش بدون شرط به ابتدای کد اصلی. ژانگ [۹۳] در پژوهش خود از این روش برای مبهم‌نگاری برنامه‌های اندرویدی استفاده کرده‌است. جابه‌جایی دستورات، روش دیگری است که در پژوهش‌های [۸۷، ۹۲، ۹۱] استفاده شده‌است. باسی و همکاران [۹۲] با استفاده از جابه‌جایی دستورات به صورت کاملاً تصادفی، با دانه^{۹۸}‌های ناهم‌گون و در نهایت استفاده از دستورات پرش، خطوط کدهای اصلی را جابه‌جا کرده و به این شکل مهاجم را در تحلیل کدهای اصلی برنامه، ناکام می‌گذارد. پردا و همکاران [۹۱] با تحلیل دالویک بایت‌کدها، بلوک‌های مستقل اجرایی در آن‌ها را استخراج کرده و ترتیب آن‌ها را عوض

Unconditional Jump^{۹۶}

Data Register^{۹۷}

Seed^{۹۸}

می‌کنند. برای تضمین اجرای منطق اصلی برنامه، از شرط‌های منطقی قطعی استفاده شده است.

استفاده از رمزگذاری در حین کامپایل برنامه‌های اندرویدی و دسترسی به اطلاعات در حین اجرا، یکی دیگر از روش‌های مبهم‌نگاری است که در سال‌های اخیر محبوبیت زیادی پیدا کرده است. روش‌های رمزگذاری را به صورت کلی می‌توان به دو دسته‌ی رمزگذاری داده و رمزگذاری منابع برنامه تقسیم کرد. در روش‌های مبتنی بر رمزگذاری داده، قسمت‌های حساسی از کد برنامه، مانند کلید دسترسی به واسط‌های برنامه‌نویسی و یا گذرواژه‌های مهم مانند آنچه برای درخواست از پایگاه داده مورد استفاده قرار می‌گیرد، رمزگذاری می‌شود. باسی و همکاران [۹۰] با استفاده از این ایده و استفاده از رمزگذاری مبتنی بر سیستم رمز سزار^{۹۹}، گذرواژه‌های حساس و حیاتی برنامه را رمزنگاری و در هنگام اجرا رمزگشایی می‌کند. منابع برنامه‌های اندرویدی، نیازمند کامپایل توسط ماشین مجازی جاوا نیستند و توسط کدهای برنامه، فراخوانی می‌شوند. پردها و همکاران [۹۱] با استفاده از چکیده‌سازی ۱۲۸ بیتی مبتنی بر MD5، فایل‌های منابع را چکیده و آن‌ها را در مسیر جدیدی مقداردهی می‌کند. از مشکلات این پژوهش آن است که نویسندگان به درستی، نحوه‌ی کارکرد کلاسی را که وظیفه‌ی بارگیری پویای فایل‌های منابع رمز شده را بر عهده دارد، مشخص نکردند. علاوه بر رمزگذاری منابع و اطلاعات حساس، رمزگذاری کلاس‌های برنامه نیز روش دیگری است که دیوید و همکاران [۹۵] از آن استفاده کرده‌اند. در این پژوهش، هر یک از کلاس‌های اندرویدی پس از رمزگذاری، در آرایه‌هایی از جنس بایت ذخیره می‌شود و کلاسی مجزا در هنگام اجرا، با استفاده از قابلیت بازتاب در برنامه‌های جاوا، کلاس رمزگذاری شده را بازیابی و اجرا می‌کند.

به صورت کلی می‌توان تاثیر پژوهش‌های مبهم‌نگاری بر برنامه‌های اندرویدی را از سه دیدگاه بررسی کرد. در حالت اول مبهم‌نگاری‌هایی نظیر تغییر نام و یا اجرای کلاس‌های رمزگذاری شده، منجر به افزایش امنیت و جلوگیری از بازبسته‌بندی می‌شود. در حالت دوم، رمزگذاری داده‌ها و یا استفاده از کدهای بومی در هنگام اجرا، مانع از اجرای تحلیل بدافزار روی برنامه مورد نظر می‌شود. در حالت پایانی، روش‌های مبهم‌نگاری نظیر چکیده‌سازی، تغییر نام و یا حذف کدهای مرده و بیهوده، منجر به افزایش کارایی و بهینه‌شدن اجرای برنامه‌های اندرویدی می‌شود. از آنجایی که مهاجمان نیز از روش‌های مبهم‌نگاری استفاده می‌کنند، بنابراین آشنایی با پژوهش‌های اخیر موجود در این حوزه می‌تواند ما را در ارائه‌ی روشی با دقت بیشتر و سرعت بالاتر یاری دهد.

Caesar Cipher^{۹۹}

۳-۵ مقایسه‌ی روش‌ها

با توجه به تحلیل‌های ارایه‌شده بر روی پژوهش‌های صورت گرفته در زمینه تشخیص برنامه‌های بازبسته‌بندی در این فصل، می‌توان گفت که اکثر پژوهش‌های موجود در این زمینه، نیازمند تقویت ایده‌ی اولیه هستند و هنوز پژوهش کاملی در این زمینه که علاوه بر دقت بالا، سرعت تشخیص مناسبی نیز داشته باشد، توسعه داده نشده‌است. اما به صورت کلی، در مقایسه‌ی روش‌های پویا و ایستا می‌توان گفت که عمده‌ی روش‌های موجود با تمرکز بر **تحلیل ایستا** سعی در افزایش کارایی روش‌های پیشین دارند. علت این موضوع آن است که تحلیل ایستا پاسخ‌گوی طیف وسیعی از برنامه‌های اندرویدی خصوصاً برنامه‌های رایگان است در حالی که پژوهش‌های موجود در زمینه‌ی تحلیل پویا، اکثراً از مجموعه‌ی داده‌ی آزمون کوچک و محدودی، خصوصاً در زمینه‌ی برنامه‌های تجاری، استفاده کرده‌اند. علت دیگر این موضوع آن است که تهدید اصلی کاربران در زمینه‌ی برنامه‌های اندرویدی بازبسته‌بندی شده، از سمت برنامه‌های رایگان رخ می‌دهد و برنامه‌های غیررایگان و تجاری، معمولاً پیش از توسعه به طور کاملی از روش‌های جلوگیری از بازبسته‌بندی استفاده می‌کنند. پژوهش پیاده‌سازی شده در این پایان‌نامه، مبتنی بر تحلیل ایستا با استفاده از گراف و استخراج ویژگی‌های مقاوم در مقابل مبهم‌نگاری است، بنابراین در ادامه‌ی این پژوهش، مقایسه‌ی روش‌های مبتنی بر تحلیل ایستا را انجام خواهیم داد. اگر به طور کلی دو معیار دقت و سرعت اجرا را در دسته روش‌های مبتنی بر تحلیل ایستا در نظر بگیریم، روش‌های مبتنی بر گراف از دقت بالایی در تشخیص برخوردار هستند چرا که تغییر گراف‌های حاصل از فراخوانی و یا اجرای برنامه سخت و هزینه‌بر است اما به صورت کلی، از آنجایی که الگوریتم‌های مقایسه‌ی گرافی، به خصوص الگوریتم‌های تشخیص گراف هم‌ریخت، سرعت بسیار پایینی در مقایسه دارند، معمولاً روش‌های این دسته به قدری کند هستند که عملاً استفاده از آن‌ها در یک ابزار کاربردی و محیط صنعتی ناممکن خواهد بود. از طرفی روش‌های مبتنی بر تحلیل ترافیک و تحلیل منابع، خصوصاً در روش‌هایی که از طبقه‌بندی استفاده می‌شود، از سرعت بالایی در تشخیص برخوردار هستند اما به جهت این‌که ایجاد مبهم‌نگاری در منابع برنامه آسان و کم‌هزینه‌است، بنابراین بازبسته‌بندی ممکن است مبتنی بر منابع برنامه نیز اتفاق بیافتد. علاوه بر این، هدف بسیاری از برنامه‌های بازبسته‌بندی، فریب کاربران ناآگاه با استفاده از ایجاد یک واسط کاربری مشابه با برنامه اصلی است، بنابراین بررسی این دسته از ویژگی‌ها می‌تواند ما را در تشخیص برنامه‌های اندرویدی بازبسته‌بندی شده یاری دهد. در جدول ۳-۲ پژوهش‌های مطرح مبتنی بر تحلیل ایستا را از نظر سه خاصیت یعنی **جست‌وجوی دودویی، سربار زمانی و دقت تشخیص** مورد مقایسه قرار داده‌ایم.

جدول ۳-۲: مقایسه‌ی روش‌های مبتنی بر تحلیل ایستا

دسته‌بندی روش‌ها	پژوهش‌کنندگان	سر بار محاسباتی	دقت تشخیص	مقایسه‌ی دودویی
مبتنی بر آپکد و دنباله‌ی دستورات	ژو و همکاران [۳۶]	متوسط	پایین	✓
	ژو و همکاران [۲۷]	متوسط	پایین	✓
	دزنور و همکاران [۳۷]	متوسط	پایین	✓
	جرومه و همکاران [۴۰]	متوسط	پایین	×
	سرنیل و همکاران [۴۱]	متوسط	پایین	×
	لین و همکاران [۴۲]	متوسط	پایین	×
مبتنی بر گراف	سان و همکاران [۹]	زیاد	خوب	✓
	هو و همکاران [۴۸]	متوسط	خوب	✓
	ژو و همکاران [۴۹]	زیاد	خوب	✓
	چن و همکاران [۵۰]	کم	خوب	✓
	جنگ و همکاران [۵۱]	زیاد	خوب	✓
	وانگ [۵۳]	زیاد	خوب	✓
	ترکی [۵۲]	زیاد	خوب	✓
	الشهری و همکاران [۵۵]	متوسط	پایین	✓
	هه و همکاران [۵۶]	متوسط	پایین	×
مبتنی بر تحلیل ترافیک شبکه	مالک و همکاران [۵۸]	متوسط	پایین	✓
	ایلند و همکاران [۵۹]	متوسط	پایین	✓
	شارما و همکاران [۶۰]	متوسط	متوسط	×
	شائو و همکاران [۱]	کم	خوب	×
	سان و همکاران [۶۲]	زیاد	خوب	✓
مبتنی بر تحلیل منابع	هو و همکاران [۶۳]	متوسط	خوب	✓
	لین و همکاران [۶۴]	متوسط	پایین	✓
	گوآ و همکاران [۶۵]	زیاد	متوسط	✓
	لیو و همکاران [۶۶]	متوسط	خوب	✓
	ما و همکاران [۶۹]	متوسط	خوب	✓

فصل ۴

راهکار پیشنهادی

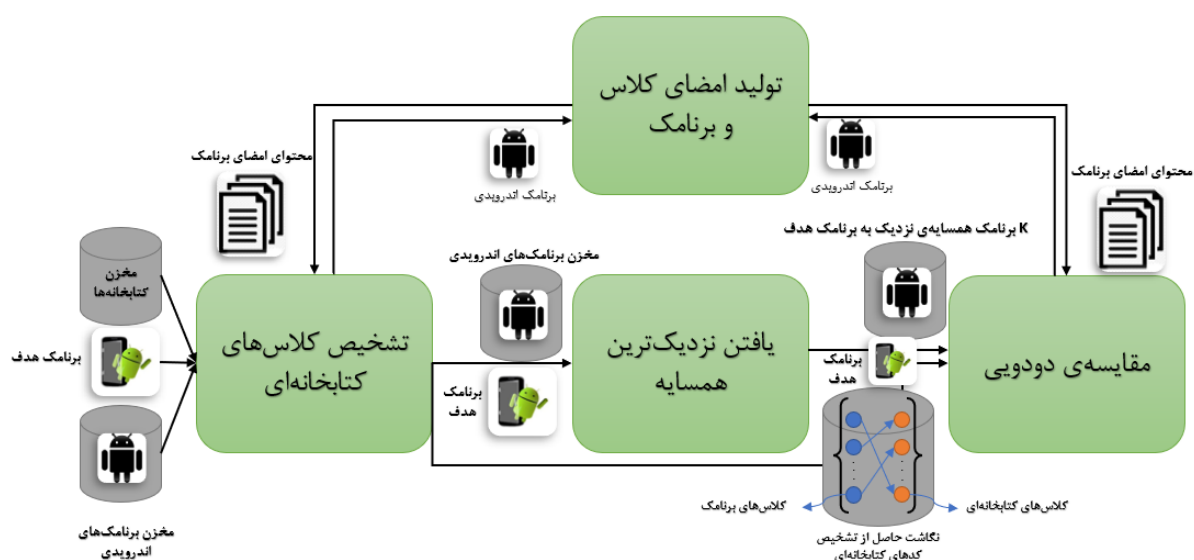
تقریباً در تمامی راهکارهای ارائه شده جهت تشخیص برنامه‌های باز بسته بندی شده، نیازمند انجام مقایسه‌ی دودویی ویژگی‌های استخراج شده از برنامه‌ها در راستای تشخیص جفت قلبی^۱ هستیم. به همین جهت اصولاً تشخیص جفت برنامه‌ها باز بسته بندی شده از میان مخزن برنامه‌ها روشی زمان‌بر و پرهزینه است. از طرفی همانطور که بررسی شد، برای تشخیص برنامه‌ها باز بسته بندی شده، نیازمند ویژگی‌هایی به صورت ایستا و یا پویا از برنامه‌ها هستیم که رفتار منطقی آن‌ها را نشان دهد. از بررسی کارهای پیشین و مطالعه در زمینه‌ی تحلیل برنامه‌های اندرویدی، می‌توان دریافت که پارامترهای دقت و سرعت در تشخیص برنامه‌های باز بسته بندی شده، رابطه‌ی مستقیمی با ویژگی‌های منتخب به جهت مقایسه‌ی دودویی برنامه‌ها دارد. پیچیدگی ویژگی‌های منتخب منجر به کاهش سرعت و مقاومت آن‌ها در مقابل روش‌های مبهم‌نگاری، در نهایت منجر به افزایش دقت در تشخیص جفت قلبی خواهد شد. در این فصل ابتدا نمای کلی از پژوهش را توضیح می‌دهیم و مولفه‌های اصلی پژوهش را به صورت مختصر بررسی خواهیم کرد. سپس هر کدام از مولفه‌های مطروحه را با جزئیات بیشتری بررسی و معایب و مزایای هر کدام را شرح خواهیم داد.

۴-۱ راهکار پیشنهادی

از آنجایی که مقایسه‌ی دودویی برنامه‌ها ورودی با تمامی برنامه‌های موجود در مخزن، هزینه‌ی محاسباتی زیادی را در روند تشخیص تحمیل می‌کند، استفاده از یک مرحله پیش پردازش^۲ می‌تواند به افزایش سرعت تشخیص کمک کند. همانطور که در شکل ۴-۱ مشخص شده است، راهکار پیشنهادی در این پژوهش به صورت کلی از سه مؤلفه‌ی «تشخیص کتابخانه‌های اندرویدی»، «یافتن نزدیک‌ترین همسایه» و «تشخیص

^۱ Repackaged Pair
^۲ Preprocess

برنامک بازبسته‌بندی شده» تشکیل شده‌است. مؤلفه‌ی «تشخیص کتابخانه‌های اندرویدی» با الهام از پژوهش [۵۲] و همراه با تغییر امضای کلاسی به جهت افزایش سرعت، پیاده‌سازی شده‌است. مخزن کتابخانه‌های اندرویدی و برنامک هدف، به عنوان ورودی به این مؤلفه داده می‌شوند. به صورت کلی در این مؤلفه کلاس‌های هربرنامک، با تمامی کلاس‌های موجود در مخزن کتابخانه‌ها مقایسه می‌شود. برای کاهش فضای مقایسه‌ی کتابخانه‌ها با کلاس‌های برنامک هدف، از «ماژول فیلتر کتابخانه‌ها» استفاده شده‌است. این ماژول تعداد کلاس‌های مورد بررسی در هر کتابخانه را با استفاده از دو «فیلتر ساختاری» و «طول امضا» کاهش می‌دهد و در نهایت با استفاده از توابع چکیده‌سازی محلی^۳، مقایسه‌ی کلاس‌های برنامک هدف و مخزن کتابخانه‌های اندرویدی در ماژول «استخراج نگاشت» انجام می‌شود. حل مسئله‌ی نگاشت میان کلاس‌های برنامک و کلاس‌های کتابخانه‌ای، همان حل مسئله‌ی تخصیص^۴ است که در آن نگاشتی از کمترین هزینه (مبتنی بر چکیده‌سازی محلی) از یک گراف دوبخشی استخراج می‌شود.



شکل ۴-۱: نمای کلی راهکار پیشنهادی

در این پژوهش از یک طبقه‌بند نزدیک‌ترین همسایه^۵ به عنوان مرحله‌ی پیش‌پردازش ورودی‌ها استفاده شده‌است. در این مؤلفه با استفاده از تعدادی ویژگی مبتنی بر واسطه‌های کاربری^۶، برنامک‌های موجود در مخزن و برنامک ورودی هدف طبقه‌بندی^۷ و تعدادی از نزدیک‌ترین همسایه‌های برنامک ورودی به عنوان برنامک‌های مشکوک به جهت بررسی دقیق‌تر به مرحله‌ی بعد خواهند رفت.

^۳ Fuzzy Hashing
^۴ Assignment Problem
^۵ Nearest neighbor
^۶ User Interface
^۷ Clustering

مؤلفه‌ی دیگری که در این پژوهش پیاده‌سازی شده است، تشخیص برنامه‌های بازبسته‌بندی شده است. این مؤلفه، در یک مرحله پیش‌پردازش با استفاده از مؤلفه‌ی تشخیص کتابخانه‌ها، نگاشتی از کلاس‌های برنامه و کلاس‌های کتابخانه‌ای مخزن به دست می‌آورد. سپس دو برنامه مورد بررسی و نگاشتی از کلاس‌های هر دو، به عنوان ورودی مازول طبقه‌بند، وارد می‌شود. در مؤلفه‌ی تشخیص برنامه‌های بازبسته‌بندی شده، ابتدا کدهای کتابخانه‌ای با استفاده از نگاشت حاصل از مرحله‌ی قبلی حذف می‌شوند و کلاس‌های باقی‌مانده برای ساخت امضا توسط مازول ساخت امضای برنامه، تحلیل می‌شوند. در نهایت با استفاده از چکیده‌سازی محلی و تعیین یک حد‌آستانه، تعیین بازبسته‌بندی صورت می‌گیرد.

۴-۲ ساخت امضای برنامه

هدف از ساخت امضای^۸ برنامه، تشکیل مدلی از رفتار منطقی و دستوری آن می‌باشد به طوری که مقاومت بالایی در مقابل راهکارهای مبهم‌نگاری داشته‌باشد. برای ساخت امضای برنامه از الحاق امضای تمامی کلاس‌های آن استفاده می‌شود. در این پژوهش، مؤلفه‌های تشخیص کتابخانه‌های اندرویدی و تشخیص بازبسته‌بندی، هر دو از امضای کلاس برای شباهت‌سنجی استفاده کرده‌اند. امضای کلاس متشکل از مهم‌ترین ویژگی‌های آن است که علاوه بر مدل‌کردن رفتار کلاس، مقاومت بالایی در مقابل روش‌های مبهم‌نگاری داشته‌باشد. برای تشخیص کدهای کتابخانه‌ای، امضای تمامی کلاس‌های جاوا در کتابخانه‌ی مذکور ساخته‌شده و برای تشخیص قسمت‌هایی که شامل کدهای کتابخانه‌ای در برنامه اندرویدی است استفاده می‌شود.

از آنجایی که امضا برای هر کلاس تولید می‌شود، بنابراین به صورت کلی دو مرحله برای تولید امضای کلاس پیاده‌سازی شده است. در ابتدا با استفاده از ویژگی‌های مبتنی بر متد^۹، امضای متدهای پویای^{۱۰} کلاس استخراج شده و سپس با استفاده از طرحی که در ادامه توضیح داده خواهد شد امضای کلاس ساخته می‌شود. در ادامه ابتدا توصیف کلی از تمامی قسمت‌های امضای کلاسی در برنامه ارائه می‌دهیم و دلیل استفاده از این پارامتر را در امضای کلاس عنوان می‌کنیم. سپس ساختار صوری^{۱۱} تشکیل امضای کلاس‌های برنامه و در نهایت امضای برنامه را شرح می‌دهیم.

Signature^۸
Method^۹
Dynamic Method^{۱۰}
Formal^{۱۱}

۴-۲-۱ توصیف پارامترهای امضای برنامه

در این بخش ویژگی‌های استفاده‌شده در امضای کلاس را شرح می‌دهیم و دلایل استفاده از آن‌ها را عنوان می‌کنیم.

- **تغییردهنده‌ی متد:** تغییردهنده‌های هر متد^{۱۲}، مشخص‌کننده‌ی چگونگی پیاده‌سازی متدهای جاوا هستند که اکثراً به سختی قابل تغییر می‌باشند چرا که در صورت تغییر آن‌ها بدنه‌ی متد^{۱۳} معمولاً باید به صورت کلی تغییر کند، بنابراین استفاده از این ویژگی، مقاومت بالایی مقابل مبهم‌نگاری را به ارمغان می‌آورد. برای ساخت امضای متد از چهار نوع تغییردهنده‌ی بومی^{۱۴}، انتزاعی^{۱۵}، ایستا^{۱۶} و سازنده^{۱۷} استفاده شده‌است. متدهای بومی، شامل دستورات به زبان بومی جاوا هستند که توسط پردازنده به صورت مستقیم اجرا می‌شود. متدهای ایستا توسط تمامی نمونه‌های کلاس قابل دسترسی هستند و تنها به متغیرهای ایستا در کلاس دسترسی دارند. متدهای سازنده، در واقع وظیفه‌ی مقداردهی اولیه به متغیرهای پویا در هنگام ساخت نمونه‌ی کلاسی^{۱۸} در برنامه‌های اندرویدی را بر عهده دارند.
- **نوع داده‌ی خروجی و ورودی متد:** ویژگی دیگری که در امضای متد مورد استفاده قرار گرفته‌است، نوع داده‌ی متغیرهای خروجی و ورودی به هر متد می‌باشد. به دلیل آن‌که تغییر ورودی و خروجی متدها غالباً بدون تغییر بدنه‌ی اصلی توابع امکان‌پذیر نیست، بنابراین تغییر آن‌ها منجر به صرف هزینه‌ی زیادی برای متقلبان می‌شود، چرا که در این صورت نیاز به تغییر بدنه‌ی متد دارند، به طوری که منطق اصلی متد حفظ شود.
- **متدهای فراخوانی‌شده:** در این قسمت از دو دسته‌ی مهم از متدهایی که در بدنه‌ی برنامه فراخوانی شده‌اند استفاده شده‌است. دسته‌ی اول متدهایی هستند که متدهای کتابخانه‌ای جاوا نیستند و به صورت غیرایستا، توسط توسعه‌دهنده پیاده‌سازی شده‌اند. لازم به ذکر است که به دلیل آن‌که در قسمت تشخیص برنامه‌های اندرویدی بازبسته‌بندی شده، کدهای کتابخانه‌ای حذف شده‌اند، بنابراین تعریف متدهای فراخوانی شده در این قسمت، بدون در نظر گرفتن متدهای کتابخانه‌ای اندرویدی خواهد بود. دسته‌ی دوم، متدهای زبان جاوا هستند که داخل بسته‌ی کتابخانه‌ای جاوا حضور دارند. تغییر این دسته از متدها آسان‌تر از متدهای قسمت قبلی است، اما در هر حال نمی‌توان بدون پرداخت هزینه‌ی محاسباتی آن‌ها را حذف کرد.

^{۱۲} Method Modifier

^{۱۳} Method Body

^{۱۴} Native

^{۱۵} Abstract

^{۱۶} Static

^{۱۷} Constructor

^{۱۸} Class Instance

- **فراخوانی واسطه‌های برنامه‌سازی:** واسطه‌های برنامه‌سازی^{۱۹}، هسته‌ی اصلی رفتار هر متد در برنامه‌های اندرویدی هستند که بدافزارنویسان نیز از آن‌ها استفاده می‌کنند. تغییر این دسته از فراخوانی‌ها در متدهای کلاسی، سخت و با پیچیدگی همراه است.

در امضای هر کلاس نیز از ویژگی‌های زیر استفاده شده‌است:

- **هسته‌ی کلاس:** هسته‌ی کلاس شامل امضای تمامی متدهای موجود با استفاده از ویژگی‌های بخش قبل می‌باشد.
- **کلاس‌های داخلی:** از آن‌جا که تغییر سلسله مراتب کلاس‌های جاوایی، اکثراً ناممکن است بنابراین امضای تمامی کلاس‌های داخلی به همراه متدهای پیاده‌سازی شده در این قسمت استفاده می‌شود. تغییر سلسله مراتب کلاس‌های داخلی نیازمند تغییر تمامی فراخوانی‌های موجود از کلاس‌های مورد نظر است بنابراین تغییر سلسله مراتب تقریباً ممکن نیست.
- **سطح کلاس:** همانطور که گفته‌شد، تغییر سلسله مراتب کلاس‌ها سخت و زمان‌بر است بنابراین شماره‌ی سطح کلاس در ساختار سلسله‌مراتبی، ویژگی مقاوم دیگری است که در این قسمت استفاده می‌شود.
- **کلاس پدر:** تغییر ساختار ارث‌بری کلاس‌ها^{۲۰}، هزینه‌بر است چرا که تغییر این ساختار نیازمند تغییر فراخوانی‌های متعدد ناشی از ساختن نمونه‌های کلاسی است. بنابراین در امضای هر کلاس، از امضای پدر آن کلاس نیز استفاده شده‌است.
- **نام کلاس داخلی و بیرونی:** برای بررسی ریزدانه‌تر^{۲۱}، علاوه بر سطح کلاس، از نام کلاس‌های درونی و بیرونی نیز استفاده شده‌است. اکثر مبهم‌نگارها، از بذر یکسانی برای تولید نام‌های تصادفی استفاده می‌کنند بنابراین ایجاد نام‌های متنوع از یک نام نیازمند تغییر نام اولیه‌ی متد است.
- **طول هسته‌ی کلاس:** طول هسته‌ی کلاس، شامل متدهای کلاسی، ویژگی دیگری است که در امضای کلاس مورد استفاده قرار گرفته‌است.

^{۱۹} Application Programming Interface

^{۲۰} Class inheritance

^{۲۱} Fine Grain

۲-۲-۴ توصیف صوری امضای متد

به صورت کلی امضای متد از ۵ قسمت متفاوت تشکیل شده است:

(۱-۴)

$\langle Modifier, RetType, InputType, JLibMethodCallee, NonStaticAppMethodCallee, ApiCallSootSignature \rangle$

- *Modifier*: در این بخش رشته‌ای شامل تغییردهنده قرار می‌گیرد. همانطور که در بخش پیشین بررسی کردیم، تغییردهنده‌ی متد، شامل یکی از انواع بومی، انتزاعی، ایستا و یا سازنده‌است. در صورتی که تغییردهنده‌ی متد هیچ‌کدام از موارد ذکر شده نباشد، مقدار *Null* به جای این ویژگی قرار می‌گیرد.

- *RetType*: نوع داده‌ی بازگشتی از متد، مرتب‌شده به صورت الفبایی، به صورت رشته در این قسمت قرار می‌گیرد. در صورتی که نوع داده‌ی متد، از انواع داده‌ای زبان جاوا نباشد، نام کلاس این نوع داده‌ای به صورت رشته جایگزین می‌شود.

- *InputType*: نوع داده ورودی‌های متد، مرتب‌شده به صورت الفبایی، به صورت رشته در این قسمت قرار می‌گیرند. در صورت نبودن نوع داده در زبان جاوا، همانند *RetType* عمل می‌شود.

- *JLibMethodCallee*: رشته‌ای متشکل از تمامی توابع کتابخانه‌ای جاوا، که در بدنه‌ی اصلی متد صدا زده شده‌اند در این قسمت قرار می‌گیرد. رشته‌ی حاصل به صورت الفبایی مرتب شده‌است.

- *NonStaticAppMethodCallee*: رشته‌ی مرتب‌شده به صورت الفبایی شامل نام تمامی توابع غیر کتابخانه‌ای و غیرایستا که توسط تابع فراخوانی شده‌اند. لازم به ذکر است، از آنجایی که فراخوانی توابع ایستا، مقاومت بالایی در مقابل مبهم‌نگاری ندارد، در این قسمت از توابع غیر ایستا استفاده شده‌است. افزودن فراخوانی توابع ایستا، یکی از روش‌های محبوب در مبهم‌نگاری برنامه‌های اندرویدی است.

- *ApiCallSootSignature*: در این قسمت رشته‌ای از سه ویژگی مهم توابع واسطه‌های برنامه‌نویسی، شامل نوع کلاس متد فراخوانی، کلاس نوع بازگشتی متد و نام متد قرار می‌گیرد. تغییر نام واسطه‌های برنامه‌نویسی قطعاً نیازمند تغییر کتابخانه‌های اندرویدی و ایجاد مبهم‌نگاری در آن‌ها است، به همین دلیل در این قسمت نام متد، تا حدودی مقاوم در مقابل مبهم‌نگاری برنامه‌ها می‌باشد.

(۲-۴)

$\langle DeclaringClass, RetType, MethodName \rangle$

۳-۲-۴ توصیف صوری امضای کلاس

امضای کلاس، شامل ۶ قسمت است که در ادامه به بررسی هر کدام از آنها می‌پردازیم:

$\langle ClassCoreSig, InnerClassesSig, InheritedClassesSig, ImpInterfacesSig$ (۳-۴)
 $, ClassLevel, InnerOuterClassName, ClassLen, NumofInnerClass \rangle$

- *ClassCoreSig*: رشته‌ی حاصل از الحاق امضای تمامی متدهای کلاس در این قسمت قرار می‌گیرد. ترتیب قرارگرفتن امضای متدهای کلاس، بر اساس خطوطی است که پیاده‌سازی شده‌اند، یعنی اگر متد *A* در کلاس *B* پیش از متد *C* پیاده‌سازی شده‌بود، آنگاه امضای متد *A* پیش از متد *C* قرار خواهد گرفت.

$\langle MethodSig_1, MethodSig_2, \dots, MethodSig_n \rangle$ (۴-۴)

- *InnerClassesSig*: رشته‌ی حاصل از الحاق امضای کلاس‌های داخلی کلاس مورد نظر، مرتب‌شده به صورت الفبایی در این قسمت قرار خواهد گرفت.

- *InheritedClassesSig*: در صورتی که کلاس پدر^{۲۲}، یکی از کلاس‌های کتابخانه‌ای جاوا باشد، نام کلاس پدر در این قسمت قرار می‌گیرد، اما در غیر این صورت، امضای کلاس پدر، شامل *CoreClassesSig* و *InnerClassesSig* به صورت رشته در این قسمت قرار می‌گیرد.

- *ImpInterfacesSig*: در صورتی که واسط پیاده‌سازی شده^{۲۳}، از نوع کلاس‌های کتابخانه‌ای جاوا باشد، نام آن به صورت رشته در این قسمت قرار خواهد گرفت و در غیر این صورت همانند قسمت قبل، امضای واسط‌ها، شامل الحاق *CoreClassesSig* و *InnerClassesSig* تمامی آن‌ها، مرتب شده به صورت الفبایی خواهد بود.

- *ClassLevel*: عمق کلاس^{۲۴} در ساختار سلسله‌مراتبی کلاس‌های هر بسته به صورت عددی در این قسمت قرار خواهد گرفت. برای مثال اگر کلاس مورد نظر، ریشه‌ی درخت سلسله‌مراتبی بسته‌ها باشد، عدد ۰ و اگر فقط یک کلاس بیرونی داشته باشد، عدد ۱ در این قسمت قرار خواهد گرفت.

- *InnerOuterClassName*: نام کلاس بیرونی و کلاس‌های درونی مرتب‌شده به صورت الفبایی به صورت رشته در این قسمت قرار خواهد گرفت.

^{۲۲} Parent Class

^{۲۳} Implemented Interface

^{۲۴} Class's Depth

- *ClassLen*: در این قسمت طول رشته‌ی امضای کلاس، به صورت عددی درج می‌شود.
- *NumOfInnerClass*: تعداد کلاس‌های داخلی کلاس هدف، به صورت عددی در این قسمت قرار می‌گیرد.

۳-۴ مؤلفه‌های تشخیص برنامه‌های بازبسته‌بندی شده

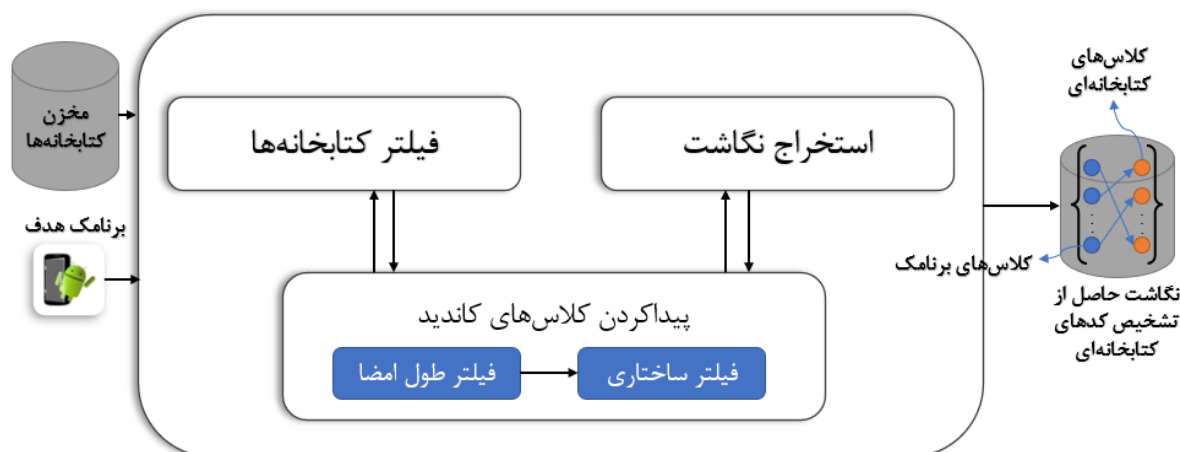
در این قسمت به بررسی ماژول^{۲۵}‌های هر کدام از سه مؤلفه‌ی تشخیص برنامه‌های بازبسته‌بندی شده می‌پردازیم.

۱-۳-۴ مؤلفه‌ی تشخیص کدهای کتابخانه‌ای

با توجه به تصویر ۴-۲، این مؤلفه دارای سه ماژول اصلی یافتن کلاس‌های کاندید^{۲۶}، فیلتر کلاس‌های کاندید مبتنی بر فیلترهای ساختاری^{۲۷} و طول امضا و در نهایت ماژول یافتن نگاشت^{۲۸} میان کلاس‌های برنامه و کلاس‌های کتابخانه‌ای است.

آسان‌ترین روش یافتن کلاس‌های کتابخانه‌ای مقایسه‌ی دودویی کلاس‌های موجود در مخزن کتابخانه‌ها با تمامی کلاس‌های برنامه می‌باشد اما این روش بسیار پرهزینه است، به همین جهت در مرحله‌ی اول از تشخیص کدهای کتابخانه‌ای با استفاده از دو فیلتر ساختاری و طولی، تعداد کلاس‌های مورد مقایسه با برنامه مقصد را کاهش می‌دهیم. در قسمت ابتدایی توسط فیلتر طولی، تنها کلاس‌هایی را بررسی خواهیم کرد که طول آن‌ها از کلاس مبدا از یک حد آستانه^{۲۹} Thr_L فاصله داشته‌باشد. در قسمت فیلتر ساختاری، با استفاده از بررسی و تحلیل چندین ویژگی روی کلاس‌های کتابخانه‌ای، مجموعه‌ی کلاس‌هایی که در خروجی فیلتر طولی ظاهر شده‌اند را مجدداً کاهش می‌دهیم. ویژگی‌های مورد بررسی در این قسمت شامل تعداد کلاس‌های بیرونی و داخلی، ارث‌بری کلاسی کلاس هدف، ارث‌بری از کلاس‌های کتابخانه‌ای جاوا و یا اندروید، تعداد واسطه‌های پیاده‌سازی شده و معادل بودن کلاس‌های بیرونی می‌باشد. کلاس خروجی پس از فیلتر طولی، تنها در صورتی بعد از اعمال فیلتر ساختاری نیز حضور خواهد داشت که در تمامی ویژگی‌های ذکر شده با کلاس هدف تطابق داشته‌باشد. شبه‌کد این رویه در الگوریتم ۱ مشخص شده‌است.

Module^{۲۵}
Candidate^{۲۶}
Structural^{۲۷}
Mapping^{۲۸}
Threshold^{۲۹}



شکل ۴-۲: نمای کلی مؤلفه‌ی تشخیص کدهای کتابخانه‌ای

در ماژول پیدا کردن کلاس‌های کاندید، تمامی کلاس‌های برنامک و یک کلاس هدف از میان کلاس‌های کتابخانه‌ای به عنوان ورودی به ماژول مورد نظر ارسال می‌شود و در نهایت لیستی از کلاس‌های برنامک به عنوان کلاس‌های مشابه در خروجی مشخص می‌شود. شبه‌کد رویه‌ی پیدا کردن کلاس‌های کاندید در الگوریتم ۲ مشخص شده‌است. پس از اجرای ماژول پیدا کردن کلاس‌های کاندید، کتابخانه‌هایی که که کلاسی مشابه در برنامک دارند را شناسایی می‌کنیم. به جهت شناسایی کتابخانه‌های موجود در برنامک، پیش از اجرای مؤلفه‌ی تشخیص کدهای کتابخانه‌ای، مخزنی شامل کتابخانه‌های اندرویدی مشهور ایجاد می‌کنیم و تمامی کلاس‌های کتابخانه‌ای را در این مخزن قرار می‌دهیم. سپس تعداد N کلاس با طول بیشینه از هر کتابخانه را انتخاب و به عنوان کلاس‌های هدف قرار می‌دهیم. در ادامه، الگوریتم پیدا کردن کلاس‌های کاندید را برای هر کدام از کلاس‌های هدف کتابخانه‌ی *lib* اجرا کرده و تعدادی کلاس کاندید از میان کلاس‌های برنامک انتخاب می‌کنیم. در این مرحله، کلاس‌های کاندید برنامک را با کلاس *Class* از کتابخانه‌ی *lib* با استفاده از روش‌های چکیده‌سازی محلی مقایسه کرده و در صورتی که میزان شباهت دو کلاس بیش از یک حد آستانه Thr_f باشد، کلاس کاندید مورد نظر با کلاس *Class* مطابقت یافته و برای نگاشت میان کلاس‌های کتابخانه‌ای و کلاس‌های برنامک، به مرحله‌ی بعد خواهد رفت.

ماژول استخراج نگاشت، به جهت ایجاد نگاشتی میان کلاس‌های کتابخانه‌ای (خروجی ماژول پیدا کردن کلاس‌های کاندید) و کلاس‌های برنامک پیاده‌سازی شده‌است. ورودی ماژول لیست Cls_{lib} شامل کلاس‌های کتابخانه‌ای برنامک، حاصل از اجرای الگوریتم ۲ و لیست Cls_{app} کلاس‌های برنامک هدف می‌باشد. خروجی ماژول، نگاشتی یک به یک و پوشا از کلاس‌های کتابخانه‌ای به کلاس‌های برنامک می‌باشد و

ورودی: کلاس هدف برای یافتن لیستی از کلاس‌های مشابه با آن t_c ، لیستی از کلاس‌های برنامه S_{app}

خروجی: لیستی از کلاس‌های کاندید برنامه مشابه با کلاس هدف t_c

۱: قرار بده $Candidate = \emptyset$

۲: قرار بده $L = length(Sig_{t_c})$ مقدار $Sig(t_c)$ امضای نهایی کلاس t_c می‌باشد

۳: \triangleleft فیلتر طولی کلاس‌های برنامه

۴: به ازای $Class \in S_{app}$:

۵: اگر $L - T_L < length(Sig_{Class}) < L + T_L$:

۶: $Candidate \cup Class$

۷: \triangleleft فیلتر ساختاری کلاس‌های برنامه

۸: به ازای $Class \in Candidate$:

۹: قرار بده $Condition = True$

۱۰: قرار بده $Condition = Condition \wedge (\#outerClass(Class) = \#outerClass(t_c))$

۱۱: قرار بده $Condition = Condition \wedge (\#InnerClass(Class) = \#InnerClass(t_c))$

۱۲: قرار بده $Condition = Condition \wedge (DoesInherit(Class) = DoesInherit(t_c))$

۱۳: قرار بده $Condition = Condition \wedge (SDKInherit(Class) = SDKInherit(t_c))$

۱۴: قرار بده $Condition = Condition \wedge (\#Interfaces(Class) = \#Interfaces(t_c))$

۱۵: قرار بده $Condition = Condition \wedge (SDKInterface(Class) = SDKInterface(t_c))$

۱۶: قرار بده $Condition = Condition \wedge (OuterClass(Class) = OuterClass(t_c))$

۱۷: اگر $Condition \neq True$:

۱۸: $Candidate = Candidate - \{Class\}$

مشخص می‌کند کدام یک از کلاس‌های برنامه، کلاس‌های کتابخانه‌ای هستند. به عبارت دیگر، خروجی ماژول استخراج نگاشت، نگاشتی به صورت زیر می‌باشد:

$$f : L \rightarrow A \quad L \subseteq Cls_{lib}, A \subseteq Cls_{app} \quad (5-4)$$

در ابتدای این ماژول، کلاس‌های کتابخانه‌ای در سطح i ام را با استفاده از متد *FilterClassByLevel* محاسبه می‌کنیم و آن را با $S_{lib,i}$ نمایش می‌دهیم. در ادامه به ازای تمامی کلاس‌های موجود در مجموعه‌ی کلاسی $S_{lib,i}$ ، کلاس‌های کاندید را با استفاده از ماژول یافتن کلاس‌های کاندید محاسبه و تحت عنوان $Candidate_{cls}$ ذخیره می‌کنیم. سپس برای محاسبه‌ی کلاس‌های منطبق^{۳۰} بر کلاس هدف، تمامی کلاس‌های کاندید حاصل از مرحله‌ی قبل را با استفاده از توابع چکیده‌سازی محلی، با کلاس هدف مقایسه و در صورتی که میزان تشابه دو کلاس از حد آستانه‌ی Thr_s بیشتر بود آنگاه کلاس مورد نظر را به عنوان کلاس تطابق داده شده به مجموعه‌ی $Matched_{class}$ اضافه می‌کنیم. در ادامه سعی می‌شود تا مسأله‌ی پیداکردن کلاس‌های

الگوریتم ۲ پیداکردن کلاس‌های کاندید

ورودی: لیست کلاس‌های برنامه S_{app} ، مخزن کتابخانه‌های $libs$

خروجی: لیست کتابخانه‌های استفاده‌شده در برنامه

۱: قرار بده $Matched = \emptyset, Libs = \emptyset$

۲: به ازای $lib \in libs$:

۳: قرار بده $GetMaxLengthClasses(lib, N) = TargetClasses$

۴: به ازای $Class \in TargetClasses$:

۵: قرار بده $Candidates = FindCandidate(Class, S_{app}), Matched_{class} = \emptyset$

۶: به ازای $Ca \in Candidates$:

۷: اگر $FHashCompare(Sig_{ca}, Sig(Class)) > Thr_F$:

۸: قرار بده $Matched_{class} = Matched_{class} \cup Ca$

۹: قرار بده $Matched = Matched \cup Matched_{class}$

۱۰: اگر $Matched \neq \emptyset$:

۱۱: قرار بده $Libs = Libs \cup lib$

۱۲: برگردان $Libs$

کتابخانه‌ای را به مسأله‌ی تخصیص در یک گراف دوبخشی (حاصل از کلاس‌های کتابخانه‌ای و کلاس‌های

^{۳۰} Matched Classes

برنامک) تبدیل کرد. به همین جهت گراف دو بخشی وزن دار G با تابع هزینه $\mathbb{N} \rightarrow C \times V$ را با استفاده از کلاس های کتابخانه ای و کلاس های برنامک تشکیل می دهیم. گراف G به صورت زیر تعریف می شود:

$$G = (U, V, E) \quad (۴-۶)$$

مجموعه ی گره های U شامل امضای کلاس های هدف کتابخانه ای در سطح i ام است و گره های V شامل تمامی کلاس های تطابق یافته با کلاس هدف cls می باشد. مجموعه یال های گراف E ، حاصل از ایجاد تطابق میان کلاس های برنامک و کلاس های کتابخانه ای و $FhashCompare$ متد مقایسه ی امضای کلاسی با استفاده از چکیده سازی محلی می باشد. متد مقایسه به ازای ورودی امضای کلاس های مورد مقایسه، عددی بین ۰ و ۱۰۰ را باز می گرداند.

$$U = \{cls \mid cls \in S_{lib,i} \wedge Matched_{cls} \neq \emptyset\} \quad (۴-۷)$$

$$V = \bigcup_{cls \in U} Matched_{cls} \quad (۴-۸)$$

$$E = \{(cls, c) \mid cls \in S_{lib,i}, c \in Matched_{cls}\} \quad (۴-۹)$$

$$C(cls, c) = 100 - FhashCompare(Sig_c, Sig_{cls}) \quad (۴-۱۰)$$

$$FhashCompare : Sig_1 \times Sig_2 \rightarrow R \quad Sig_1, Sig_2 \in ClassesSigs, 0 \leq R \leq 100 \quad (۴-۱۱)$$

گراف G یک گراف دو بخشی حاصل از گره های U و V می باشد که توسط مجموعه یال های E به یکدیگر متصل شده اند. برای نگاشت گره های مجموعه ی V به مجموعه ی U از حل مسئله ی تخصیص برای سطح i ، در گراف G استفاده شده است. خروجی ماژول نگاشت کتابخانه ها، تابع یک به یک و پوشا خواهد بود. در نهایت اجتماع توابع یک به یک و پوشای به دست آمده به ازای هر یک از کلاس های کتابخانه ای، خروجی ماژول نگاشت می باشد که همان نگاشت میان کلاس های کتابخانه ای و کلاس های برنامک می باشد. شبکه کد رویه ی نگاشت میان کلاس های کتابخانه ای و کلاس های برنامک را می توان در الگوریتم ۳ مشاهده نمود.

الگوریتم ۳ نگاشت کلاس‌های کتابخانه و برنامه

ورودی: لیست کلاس‌های کتابخانه‌ای برنامه شامل Cls_{lib} ، لیست کلاس‌های برنامه شامل Cls_{apps}

خروجی: لیستی از دوتایی‌های $(LibraryClass, AppClass)$

۱: قرار بده $f = \emptyset, i = \emptyset$

۲: تا وقتی $Cls_{lib} \neq \emptyset$:

۳: قرار بده $S_{lib,i} = FilterClassByLevel(S_{lib}, i)$

۴: قرار بده $S_{lib} = S_{lib} - S_{lib,i}$

۵: قرار بده $U = \emptyset, V = \emptyset, E = \emptyset$

۶: به ازای $cls \in S_{lib,i}$:

۷: قرار بده $Candidate_{cls} = FindCandidates(cls, Cls_{app})$

۸: قرار بده $M_{cls} = \emptyset$

۹: به ازای $c \in Candidate_{cls}$:

۱۰: اگر $FhashCompare(Sig_c, Sig_{cls}) > Thr_s$

۱۱: قرار بده $Matched_{cls} = Matched_{cls} \cup c$

۱۲: قرار بده $E = E \cup (c, cls)$

۱۳: اگر $M_{cls} \neq \emptyset$:

۱۴: قرار بده $U = U \cup \{cls\}$

۱۵: قرار بده $V = V \cup M_{cls}$

۱۶: قرار بده $f_i = AssignmentSolver(U, V, E)$

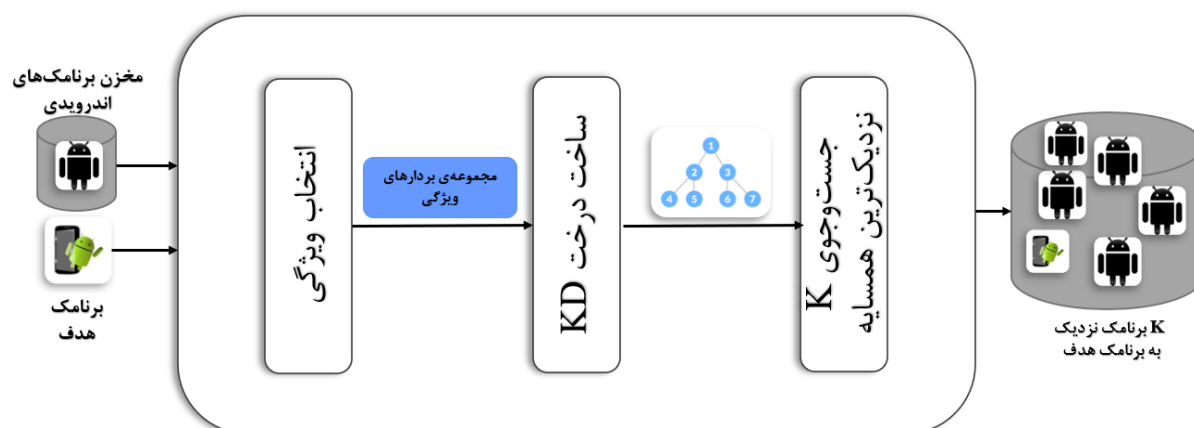
۱۷: قرار بده $f = f \cup f_i$

۱۸: قرار بده $i = i + 1$

۱۹: برگردان f

۲-۳-۴ مؤلفه‌ی یافتن نزدیک‌ترین همسایه

جهت افزایش سرعت تشخیص برنامه‌های بازسته‌بندی شده، از یک مرحله‌ی پیش‌پردازش شامل الگوریتم k نزدیک‌ترین همسایه^{۳۱} استفاده شده‌است. همانطور که در شکل ۳-۴ مشاهده می‌شود به جهت اجرای الگوریتم نزدیک‌ترین همسایه، ابتدا ۵ ویژگی، مبتنی بر منابع برنامه با استفاده از ماژول استخراج ویژگی، با استفاده از الگوریتم ۴ استخراج می‌شود و درخت ۵ بعدی حاصل از اجرای الگوریتم ۵ ساخته می‌شود. در نهایت با استفاده از اجرای رویه‌ی یافتن نزدیک‌ترین همسایه‌ی در الگوریتم ؟؟، برنامه‌های کاندید جهت مقایسه‌ی دودویی را انتخاب می‌کنیم.



شکل ۳-۴: نمای کلی مؤلفه‌ی یافتن نزدیک‌ترین همسایه

انتخاب ویژگی

به جهت اجرای فرایند یافتن نزدیک‌ترین همسایه‌های هر برنامه، نیازمند مجموعه‌ای از ویژگی‌های هر برنامه هستیم که مقاومت بالایی مقابل راهکارهای مبهم‌نگاری داشته‌باشد. از آنجا که در قسمت مقایسه‌ی دودویی و تشکیل امضای برنامه، بیشتر از ویژگی‌های مبتنی بر کد برنامه‌های اندرویدی استفاده شده‌است، در این قسمت برای تمایز بهتر برنامه‌ها از ویژگی‌های مبتنی بر منابع برنامه که واسطه کاربری آن‌را تشکیل می‌دهد استفاده کرده‌ایم. در ادامه به بررسی و توضیح دلیل انتخاب هر کدام از ویژگی‌های استخراج‌شده در این قسمت می‌پردازیم.

- **تعداد فعالیت‌ها:** هر فعالیت^{۳۲} یک صفحه‌ی جداگانه را در برنامه‌های اندرویدی به منظور پیاده‌سازی واسطه کاربری در اختیار توسعه‌دهندگان قرار می‌دهد. از آنجایی که پیاده‌سازی فعالیت‌های جدید،

^{۳۱}K-Nearest Neighbors
^{۳۲}Activity

نیازمند افزودن آن‌ها در حلقه‌ی فراخوانی فعالیت‌هاست، بنابراین، برنامه‌های بازبسته‌بندی شده عموماً دارای تعداد فعالیت مشخص و مشابه هستند چرا که بخشی از اهداف آن‌ها در صورتی محقق می‌شود که کاربران به جهت شباهت‌های واسط کاربری دچار خطا شوند. برای استخراج فعالیت‌های هر برنامه، از فایل *AndroidManifest.xml* استفاده شده‌است. همانطور که در الگوریتم ۴ مشاهده می‌شود، فعالیت‌های اندرویدی در فایل *AndroidManifest.xml* با تگ مشخص *Activity* مشخص شده‌اند. نمونه‌ای از تعریف یک فعالیت در تصویر ۴-۴ مشاهده می‌شود.

```
<manifest ... >
  <application ... >
    <activity android:name=".ExampleActivity" />
    ...
  </application ... >
  ...
</manifest >
```

شکل ۴-۴: نمونه‌ای از تعریف یک فعالیت

• **تعداد مجوزهای دسترسی برنامه:** ویژگی دیگری که برای طبقه‌بندی از آن استفاده شده‌است، تعداد مجوزهای دسترسی^{۳۳} هر برنامه است که در طی اجرای برنامه از کاربر درخواست می‌شود. از آنجایی که اجرای بدافزارهای بازبسته‌بندی شده، نیازمند دسترسی‌های گوناگون است بنابراین بدافزارنویسان برای اهداف خود از مجوزهای دسترسی به کرات استفاده می‌کنند. همانند تعداد فعالیت‌های برنامه، برای شمارش تعداد مجوزهای دسترسی، نیازمند شمارش تگ مخصوص *uses – permission* در فایل فراداده‌ی^{۳۴} *AndroidManifest.xml* هستیم. نمونه‌ای از تعریف حق دسترسی جهت استفاده از دوربین کاربر را در تصویر ۴-۵ مشاهده می‌کنید.

```
<manifest ...>
  <uses-permission android:name="android.permission.CAMERA"/>
  <application ...>
    ...
  </application>
</manifest>
```

شکل ۴-۵: نمونه‌ای از تعریف یک حق دسترسی

^{۳۳}permissions
^{۳۴}Meta-Data

- **تعداد فیلترهای تصمیم:** جابه‌جایی میان صفحات برنامه و یا جابه‌جایی میان صفحات برنامه‌های اندرویدی، با استفاده از فیلترهای تصمیم^{۳۵} صورت می‌گیرد. با استفاده از فیلترهای تصمیم برنامه مولفه‌ی^{۳۶} درخواستی خود را به سیستم عامل اعلام می‌کند. بنابراین رفتار برنامه‌های اندرویدی، از جهتی مبتنی بر فیلترهای تصمیم هستند چرا که عمده‌ی فعالیت‌های یک برنامه مبتنی بر جابه‌جایی میان صفحات مختلف (فعالیت‌ها) با استفاده از فیلترهای تصمیم می‌باشد. در این قسمت تمامی فیلترهای تصمیم ضمنی و یا صریح، با استفاده از فایل *AndroidManifest.xml* و تگ مخصوص *intent – filter* استخراج می‌شود.

- **میانگین تعداد فایل‌های png:** برای محاسبه‌ی میانگین تعداد فایل‌های تصویری که پسوند *png* دارند، از شمارش تعداد پوشه‌های *drawable* که حاوی تصاویر برنامه هستند استفاده شده است. منظور از میانگین در این ویژگی، محاسبه‌ی تقسیم تعداد تمامی فایل‌های *png* بر تعداد پوشه‌های *drawable* می‌باشد.

- **میانگین تعداد فایل‌های xml:** همانطور که در الگوریتم ۴ مشاهده می‌شود، برای محاسبه‌ی میانگین تعداد فایل‌های با پسوند *xml* از محاسبه‌ی تعداد پوشه‌های موجود در پوشه‌ی منابع برنامه یعنی */res* استفاده شده است. از آنجایی که عناصر موجود در واسط کاربری برنامه توسط فایل‌های *xml* پیاده‌سازی می‌شود، بنابراین حذف و یا اضافه کردن آن‌ها نیازمند صرف هزینه می‌باشد. دلیل استفاده از میانگین در دو ویژگی اخیر، ناشی از عدم تغییر این عدد به ازای اضافه کردن پوشه‌های جدید است. برای مثال در صورتی که یک برنامه با استفاده از یک زبان دوم بازبسته‌بندی شود، یک پوشه شامل منابع آن ایجاد می‌شود. حال در این صورت تعداد کل منابع *xml* اضافه شده است اما میانگین آن‌ها تغییر چندانی نمی‌کند.

پس از جمع‌آوری ویژگی‌های ذکر شده، با فرض آن‌که $\vartheta = \{\alpha^i | i \in [1..N]\}$ مجموعه‌ی برنامه‌های موجود در مخزن باشد، بردار ویژگی هر برنامه را با $Features_i = (f^i_1, \dots, f^i_5)$ که به ترتیب شامل تعداد فعالیت‌ها، تعداد دسترسی‌های درخواستی، تعداد فیلترهای تصمیم، میانگین تعداد فایل‌های *xml* و میانگین تعداد فایل‌های *png* نشان می‌دهیم. پس از ایجاد بردار ویژگی $Features_i$ ، به منظور هم‌سانی داده‌های آزمون، نرمال‌سازی با استفاده از تابع *Normalized* روی داده‌ها صورت می‌گیرد. معادله‌ی تابع *Normalized* به ازای هر ویژگی f^i_j زیر مشخص می‌شود.

Intent Filters^{۳۵}
Component^{۳۶}

الگوریتم ۴ استخراج ویژگی‌های مبتنی بر منابع

ورودی: برنامه‌های موجود در مخزن $DsApps$ و فایل فراداده‌ی هر کدام $Manifest_{app}$

خروجی: ویژگی‌های ۵ تایی از هر برنامه شامل مجموعه‌ی $Features$

۱: قرار بده $Features = \emptyset$

۲: به ازای $app \in DsApps$:

۳: به ازای $tag \in Manifest_{app}$:

۴: اگر $tag = Activity$:

۵: $Features_{app,AcNumber} = Features_{app,AcNumber} + ۱$ قرار بده

۶: اگر $tag = uses - premission$:

۷: $Features_{app,PNumber} = Features_{app,PNumber} + ۱$ قرار بده

۸: اگر $tag = intent - filter$:

۹: $Features_{app,IfNumber} = Features_{app,IfNumber} + ۱$ قرار بده

۱۰: قرار بده $\#TotalDrawableDir = GetTotalDrNum()$

۱۱: قرار بده $\#TotalPngFiles = GetTotalPngs("/res")$

۱۲: قرار بده $Features_{app,AvgPngPerDir} = \frac{\#TotalPngFiles}{\#GetTotalDrNum}$

۱۳: قرار بده $\#TotalResDir = GetTotalDirNum()$

۱۴: قرار بده $\#TotalXmlFiles = GetTotalXmles("/res")$

۱۵: قرار بده $Features_{app,AvgXmlPerDir} = \frac{\#TotalXmlFiles}{\#GetTotalDirNum}$

۱۶: برگردان $Features$

$$Normalized(f_j^i) = \sqrt{\frac{f_j^i - \min(f_j^1, \dots, f_j^N)}{\max(f_j^1, \dots, f_j^N) - \min(f_j^1, \dots, f_j^N)}} \quad (۱۲-۴)$$

ماژول استخراج ویژگی‌های مبتنی بر منابع به عنوان ورودی برنامه‌های موجود در مخزن و فایل فراداده‌ی *AndroidManifest.xml* و به عنوان خروجی، لیستی از ویژگی‌های ۵ تایی ذکر شده برای هر برنامه باز می‌گرداند.

ساخت درخت KD

در ادامه برای طبقه‌بندی داده‌های هر برنامه از یک طبقه‌بند مبتنی بر نزدیک‌ترین همسایه استفاده شده است. به صورت کلی دو روش پیاده‌سازی برای جست‌وجوی نزدیک‌ترین همسایه وجود دارد. در بدیهی‌ترین حالت ابتدا فاصله‌ی یک نقطه از تمامی نقاط دیگر را محاسبه، در ادامه با استفاده از مرتب‌سازی مبتنی بر فاصله، تعدادی از نزدیک‌ترین همسایه‌ها را استخراج می‌کنیم. همانطور که مشخص است روش مورد نظر نیازمند محاسبه‌ی فاصله‌ی نقطه‌ی ورودی با تمامی نقاط دیگر است. در روش دیگری که به *KD-KNN* شهرت دارد، ابتدا تمامی داده‌های برنامه را در یک داده‌ساختار^{۳۷} درختی ذخیره و برای جست‌وجوی *K* تا از نزدیک‌ترین همسایه‌های هر گره، از آن استفاده می‌کنیم. در این ساختار درختی برای جست‌وجوی گره‌های نزدیک، تمامی داده‌های آزمون به بلوک‌های نزدیک به هم تقسیم و پردازش گره‌های نزدیک تنها در این بلاک صورت می‌گیرد. در ادامه برای درک بهتر الگوریتم جست‌وجوی درختی *KNN* مثالی از یک مجموعه داده‌ی آزمون دو بعدی را در الگوریتم ۵ بررسی خواهیم کرد.

پس از اجرای الگوریتم ابتدا نقاط ورودی بر اساس عنصر اول دوتایی‌های ورودی جداسازی و مرتب می‌شوند. در ادامه میانه‌ی^{۳۸} داده‌های مرتب‌شده‌ی حاصل از مرحله‌ی قبل، محاسبه و عناصر داده‌ای بیش از مقدار میانه در سمت راست درخت و عناصر کم‌تر در سمت چپ درخت تقسیم می‌شوند. این مرحله، در واقع سازوکار ایجاد بلوک‌های داده‌ای انجام می‌شود. از آنجایی که فاصله‌ی دو نقطه در فضا از یکدیگر با استفاده از معادله‌ی ۴-۱۳ محاسبه می‌شود، بنابراین جداسازی بلوک‌های داده‌ای در الگوریتم ۵ به صورت یکی در میان، بر اساس عناصر اول و دوم از لیست داده‌ای تقسیم می‌شوند.

$$d(x, y) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (۱۳-۴)$$

برای ساخت درخت مبتنی بر ویژگی‌های استخراج شده از الگوریتم ۴، از بردار ویژگی حاصل استفاده کرده و در نهایت درخت ۵ بعدی حاوی اطلاعات برنامه‌های اندرویدی ساخته می‌شود. هر گره درخت

^{۳۷}Data Structure
^{۳۸}Median

مذکور حاوی یک بردار ویژگی ۵ تایی است و مطابق الگوریتم ۵ تصمیم‌گیری در مورد تقسیم داده‌های موجود در هر سطح با توجه به یکی از عناصر ۵ تایی صورت می‌گیرد. به عنوان مثال، زمانی که در سطح اول و ریشه‌ی درخت هستیم، عناصر اول ۵ تایی‌های مجموعه داده را جدا و آن‌ها را مرتب می‌کنیم. سپس با توجه به محاسبه‌ی میانه مقدار $Median$ را برای این گروه داده‌ای محاسبه کرده و ۵ تایی مربوط به عنصر $Median$ در ریشه قرار می‌گیرد. سپس باقی مجموعه‌ی داده‌ای با توجه به بزرگ‌تر بودن و یا کوچک‌تر بودن از عنصر $axis$ که همان شاخص تصمیم است، تقسیم می‌شوند و این رویه به صورت بازگشتی برای تمامی بلوک‌های داده‌ای و گره‌های درخت تکرار می‌شود تا زمانی که درخت ساخته شود.

ماژول ساخت درخت KD در این پژوهش لیستی از عناصر ۵ تایی حاوی ویژگی‌های مبتنی بر منابع و به طول n می‌باشد. خروجی ماژول یک درخت ۵ بعدی حاوی ویژگی‌های مذکور است که می‌توان رویه‌ی جست‌وجوی نزدیک‌ترین همسایه مطابق الگوریتم ۲۲ را از مرتبه‌ی $\log(n)$ اجرا کرد.

الگوریتم ۵ ساخت درخت دوبعدی جست‌وجوی K نزدیک‌ترین همسایه

ورودی: لیستی از داده‌های دوتایی $Points$ به صورت (x, y) و به طول n

ورودی: عمق سطح $Depth$

خروجی: درخت دوبعدی شامل داده‌های ورودی

۱: قرار بده $Dimension = Depth \bmod 2$

۲: اگر $Dimension = 1$:

۳: قرار بده $Median = GetMedian(Sort(x_1, \dots, x_n))$

۴: در غیر این صورت:

۵: قرار بده $Median = GetMedian(Sort(y_1, \dots, y_n))$

۶: قرار بده $Node.data = Median$

۷: قرار بده $Node.axis = Dimension$

۸: قرار بده $Node.leftChild = kdtree(points \text{ in } points \text{ before } median, depth + 1)$

۹: قرار بده $Node.rightChild = kdtree(points \text{ in } points \text{ after } median, depth + 1)$

جست‌وجوی k همسایه‌ی نزدیک

پس از ساخت داده‌ساختار درختی پنج‌بعدی در مرحله‌ی قبل، حال هدف یافتن نزدیک‌ترین همسایه‌های هر برنامه‌ک به جهت بررسی دوتایی آن‌ها می‌باشد. به همین جهت در این قسمت از الگوریتم جست‌وجوی

نزدیک‌ترین همسایه که در پژوهش [۹۶] معرفی شده است، استفاده کرده‌ایم. شبه‌کد این رویه را می‌توان در الگوریتم ۶ مشاهده نمود.

الگوریتم ۶ جست‌وجوی نزدیک‌ترین همسایه

- ورودی: ریشه‌ی درخت $Root$
- ورودی: نقطه‌ی آزمون $TPoint$
- خروجی: نزدیک‌ترین نقطه به ورودی $NearestPoint$
- ۱: $Path = \emptyset$ قرار بده
 - ۲: تا وقتی $Root \neq leaf$:
 - ۳: $Path.add(Root)$
 - ۴: اگر $Root[data][Root.axis] > TPoint[axis]$
 - ۵: $Searchtree(rightChild)$
 - ۶: در غیر این صورت:
 - ۷: $Searchtree(leftChild)$
 - ۸: به ازای $Point \in Path$:
 - ۹: $NearestDist = Distance(Point, TPoint)$ قرار بده
 - ۱۰: اگر $|Point[Data][axis] - TPoint[axis]| > |TPoint[axis] - Root[Data][axis]|$:
 - ۱۱: $Travel(Root.nextchild)$
 - ۱۲: $Dist = Distance(ChildPoint, TPoint)$ قرار بده
 - ۱۳: اگر $Dist > NearestDist$:
 - ۱۴: $NearestDist = Dist$ قرار بده
 - ۱۵: $NPoint = Point$ قرار بده
 - ۱۶: $NearestPoint = NPoint$ قرار بده
 - ۱۷: برگردان $NearestPoint$
-

ورودی الگوریتم شامل ریشه‌ی درخت KD و هدف آن یافتن نزدیک‌ترین نقطه از میان نقاط درون درخت به نقطه‌ی ورودی $Tpoint$ می‌باشد. به طور کلی الگوریتم مذکور را می‌توان به دو قسمت تقسیم کرد، در قسمت اول در طی یک روند رو به جلو سعی می‌کنیم با توجه به نقطه‌ی ورودی، مکان درست

آن را در برگ درخت بیابیم. به همین منظور از یک ساختار مبتنی بر درخت تصمیم^{۳۹} استفاده می‌کنیم. در این ساختار، در صورتی که مقدار شاخص تصمیم^{۴۰} *axis* بردار ورودی، از همین مقدار در بردار گره بیشتر باشد، آن‌گاه جست‌وجو را در زیردرخت سمت راست ریشه انجام می‌دهیم و در صورتی که مقدار آن کوچکتر باشد آن‌گاه زیر درخت سمت چپ را بررسی می‌کنیم. این روند بازگشتی را تا زمانی که به گره برگ برسیم ادامه می‌دهیم و مسیر طی شده را ذخیره می‌کنیم. در مرحله‌ی بعدی با توجه به نقاط درون مسیر *Path* تمامی گره‌های مسیر را به صورت عقب‌گرد^{۴۱} بررسی و نزدیک‌ترین بردار ویژگی^{۴۲} به گره ورودی را پیدا کرده و آن را باز می‌گردانیم. متد *Distance* در الگوریتم ۶ فاصله‌ی میان دو بردار ۵ تایی را با استفاده از فاصله‌ی اقلیدوسی به صورت زیر محاسبه می‌کند.

$$d(f^1, f^2) = \sqrt{(f_1^1 - f_1^2)^2 + (f_2^1 - f_2^2)^2 + \dots + (f_5^1 - f_5^2)^2} \quad (14-4)$$

پس از ایجاد درخت و پیاده‌سازی الگوریتم نزدیک‌ترین همسایه، تعداد *K* فراخوانی از الگوریتم ۶ منجر به دریافت لیستی از *K* نزدیک‌ترین همسایه‌های بردار ورودی می‌شود

۳-۳-۴ مقایسه‌ی دودویی و تشخیص برنامه‌های بازبسته‌بندی شده

هدف از پیاده‌سازی مرحله‌ی طبقه‌بندی، کاهش فضای مقایسه‌ی دودویی در این مرحله بوده‌است که منجر به افزایش سرعت مقایسه می‌شود. پس از اجرای الگوریتم یافتن *K* تا از نزدیک‌ترین همسایه‌های برنامه ورودی با استفاده از ویژگی‌های مبتنی بر منبع، در این قسمت مقایسه‌ی دودویی میان *K* برنامه و برنامه ورودی صورت می‌گیرد.

پس از شناسایی کلاس‌های کتابخانه‌ای در الگوریتم ۳، نگاشتی از کلاس‌های هر برنامه داخل مخزن و برنامه هدف با کلاس‌های کتابخانه‌ای به صورت $F : LibraryClasses \rightarrow AppClasses$ به دست می‌آید. در ادامه امضای هر کدام از کلاس‌های برنامه را تشکیل می‌دهیم. امضای هر برنامه از الحاق تمامی امضای کلاس‌های آن ایجاد می‌شود. سپس شروع به حذف کلاس‌های کتابخانه‌ای با توجه به نگاشت *F* از هر برنامه می‌کنیم و مجموعه‌ی کلاس‌های برنامه با حذف کتابخانه‌های اندرویدی تشکیل مجموعه‌ی S'_{app} را می‌دهند که به صورت زیر محاسبه شده‌است. *AppClasses* مجموعه‌ی تمامی کلاس‌های کتابخانه‌ای است که در نگاشت *F* به دست آمده‌است.

$$Cls'_{app} = Cls_{app} - AppClasses \quad (15-4)$$

Decision Tree^{۳۹}
Decision Index^{۴۰}
Back Track^{۴۱}
Feature Vector^{۴۲}

پس از حذف کلاس‌های کتابخانه‌ای از هر امضا، در این مرحله امضای اصلی برنامه‌های حاوی کلاس‌هایی که توسط توسعه‌دهنده پیاده‌سازی شده‌اند را با یکدیگر مقایسه می‌کنیم. مقایسه‌ی امضای برنامه‌ها با استفاده از روش‌های چکیده‌سازی محلی صورت می‌گیرد. استفاده از روش‌های چکیده‌سازی معمولی، نظیر MD5 برای تشابه‌سنجی، منجر به افزایش خطا در صورت مبهم‌نگاری برنامه‌ها خواهد شد چرا که روش‌های معمول عموماً برای تولید شناسه‌ی یکتا^{۴۳} کاربرد دارند و در صورتی که قسمت کوچکی از ورودی آن‌ها تغییر کند، آنگاه چکیده‌ی جدید حاصل از این توابع، به صورت کامل متفاوت خواهد بود. بنابراین استفاده از این روش‌ها برای شباهت‌سنجی میان امضای برنامه‌های اندرویدی پیشنهاد نمی‌شود. به همین منظور، در شباهت‌سنجی میان فایل‌های متنی و یا به جهت تشخیص تکرار ساختارهای واحد در فایل‌های مشابه، از روش‌های مبتنی بر چکیده‌سازی محلی استفاده می‌شود. ساختار کلی این روش‌ها، استفاده از تشابه میان بلوک‌های تکراری در متون می‌باشد و به همین دلیل، اگر دو فایل مشابه به عنوان ورودی به آن‌ها داده‌شود، آنگاه چکیده‌ی نهایی نیز به همان میزان مشابه خواهد بود. در صورتی که میزان تشابه میان امضای نهایی برنامه‌های موجود در مخزن از یک حد آستانه مانند Thr_c کمتر باشد، آنگاه دو برنامه مورد نظر بازبسته‌بندی شده تشخیص داده می‌شوند.

فصل ۵

ارزیابی

روش پیاده‌سازی شده در این پژوهش را می‌توان از نظر ساختار آن، در دسته‌ی پژوهش‌های مبتنی بر تحلیل ایستا قرار داد. یکی از آخرین پژوهش‌های موجود در این دسته، روشی است که توسط آقای ترکی [۵۲] مبتنی بر تحلیل گراف فراخوانی کلاسی^۱ و استخراج امضا از کلاس‌های برنامه‌ک ارائه شده‌است. پژوهش آقای ترکی [۵۲] در ادامه‌ی پژوهش آقای وانگ [۵۳] و به جهت بهبود دقت آن انجام شده‌است. از آنجایی که هدف این پژوهش افزایش کارایی پژوهش‌های مرتبط با تشخیص برنامه‌های بازبسته‌بندی شده می‌باشد، بنابراین در این فصل روش پیشنهادی را از دو منظر سرعت تشخیص و معیارهای دقت^۲ و فراخوان^۳ بررسی خواهیم کرد و قسمتی از پژوهش را به مقایسه‌ی راهکار پیشنهادی این پژوهش با آخرین روش مشابه اختصاص خواهیم داد. در ادامه‌ی ارزیابی، معیارهای دقت و فراخوان را به صورت زیر تعریف می‌کنیم. منظور از TP, FP, FN به ترتیب تعداد منفی غلط^۴، مثبت غلط^۵ و مثبت صحیح^۶، می‌باشد. منظور از منفی غلط، حالتی است که در آن جفت بازبسته‌بندی شده به اشتباه تشخیص داده نمی‌شود. مثبت غلط، حاصل از تشخیص بازبسته‌بندی برای یک جفت بازبسته‌بندی نشده و مثبت صحیح، زمانی است که ما جفت بازبسته‌بندی شده را به درستی تشخیص می‌دهیم.

$$Precision = \frac{TP}{TP + FP} \quad (۱-۵)$$

$$Recall = \frac{TP}{TP + FN} \quad (۲-۵)$$

Class Call Graph^۱
Precision^۲
Recall^۳
False Negative^۴
False Positive^۵
True Positive^۶

در ادامه‌ی این فصل ابتدا پارامترهای ثابت موجود در الگوریتم‌های ارائه‌شده در این پژوهش را عنوان و مقدار آن‌ها را به ازای ارزیابی‌های انجام‌شده، اعلام می‌کنیم. سپس مجموعه‌ی داده‌ای مورد ارزیابی در این پژوهش و ویژگی‌های آن را بررسی خواهیم کرد. سپس ارزیابی راهکار پیشنهادی پژوهش را از دو دیدگاه انجام می‌دهیم. در قسمت اول، ابتدا معیارهای ارزیابی را در مقایسه‌ی دودویی برنامه‌های اندرویدی، بر روی مجموعه‌ی داده^۷ بررسی می‌کنیم. اجرای ارزیابی با توجه به ایده‌ی مطروحه در مرحله‌ی تشکیل امضا و مقایسه‌ی آن با پژوهش پیشین انجام می‌گیرد. در قسمت دوم، ارزیابی پژوهش با استفاده از افزودن الگوریتم طبقه‌بندی نزدیک‌ترین همسایه انجام می‌گیرد و مقایسه‌ی دودویی با استفاده از کاهش فضای مقایسه، به عنوان ایده‌ی اصلی این پژوهش مطرح می‌گردد.

۱-۵ پارامترهای پژوهش

در این قسمت پارامترهای ثابت استفاده شده در رویه‌های مشروح در فصل ۴ را بررسی و مقدار هر کدام را برای اجرای ارزیابی پژوهش را عنوان می‌کنیم.

۱-۱-۵ مولفه‌ی تشخیص کتابخانه‌های اندرویدی

برای تحلیل ایستای برنامه‌های اندرویدی و تشکیل امضای کلاسی، از چارچوب سوت استفاده شده است. چارچوب سوت ابزاری مبتنی بر زبان جاوا است که اجازه‌ی تحلیل برنامه‌های اندرویدی و استخراج ویژگی‌های گوناگون کدپایه^۸ از آن را به کاربران می‌دهد. در قسمت تشکیل امضای کلاس‌های کتابخانه‌ای و استخراج ویژگی‌های مذکور در فصل ۴ از این ابزار استفاده شده است.

در قسمت مقایسه‌ی امضای کلاسی به منظور یافتن کلاس‌های کاندید و تشکیل نگاشت میان کلاس‌های کتابخانه‌ای و کلاس‌های برنامه و همچنین مقایسه‌ی امضای نهایی برنامه‌های اندرویدی، از روش‌های چکیده‌سازی محلی استفاده شده است. به جهت ارزیابی روش پژوهش، از سه شمای^۹ چکیده‌سازی محلی، شامل روش‌های *TLSH*, *SSdeep* و *Sdhash* استفاده شده است. در قسمت ارزیابی مقایسه‌ی دودویی، به تفصیل به مقایسه‌ی دقت و زمان محاسبه‌ی چکیده در این سه روش خواهیم پرداخت.

در ادامه، پارامترهای ثابت استفاده‌شده در مؤلفه‌ی تشخیص کدهای کتابخانه‌ای را بررسی و مقدار ثابت آن‌ها را توضیح می‌دهیم.

Data Set^۷
Code Base^۸
schema^۹

- پارامتر Thr_L : پارامتر مذکور به عنوان پارامتر ثابت در مازول فیلتر طول امضا، به جهت فیلتر کلاس‌های کاندید مبتنی بر طول کلاس هدف استفاده شده است. مقدار این پارامتر بر اساس طول کلاس هدف و بر اساس ضریبی از آن محاسبه می‌شود که روال محاسبه‌ی آن در الگوریتم ۷ مشاهده می‌شود.

الگوریتم ۷ محاسبه‌ی T_L

ورودی: کلاس هدف در فیلتر طولی T_C

خروجی: مقدار T_L مبتنی بر طول امضای کلاس هدف

۱: قرار بده $L = length(Sig_{T_C})$

۲: اگر $L \leq 1000$:

۳: $T_L = L \times 0.6$

۴: در غیر این صورت: اگر $L \leq 5000$:

۵: $T_L = L \times 0.5$

۶: در غیر این صورت: اگر $L \leq 10000$:

۷: $T_L = L \times 0.4$

۸: در غیر این صورت: اگر $L \leq 50000$:

۹: $T_L = L \times 0.3$

۱۰: در غیر این صورت: اگر $L \leq 100000$:

۱۱: $T_L = L \times 0.2$

۱۲: در غیر این صورت:

۱۳: $T_L = L \times 0.2$

۱۴: برگردان T_L

- پارامتر Thr_F : مقدار این پارامتر در مازول فیلتر کتابخانه‌ها به عنوان حد کمینه برای تشابه میان کلاس‌های کتابخانه‌ای و برنامه‌ک استفاده می‌شود که به جهت ارزیابی در این پژوهش مقدار ۷۰ گرفته است.

- پارامتر Thr_S : این پارامتر به جهت حد کمینه‌ی تشابه میان کلاس‌های کتابخانه‌ای و برنامه‌ک به منظور ایجاد نگاشت استفاده می‌شود که مقدار آن برابر ۵۰ قرار داده شده است.

- پارامتر N : مقدار این پارامتر در مازول فیلتر کتابخانه‌ها، برای تعیین تعداد کلاس‌های کتابخانه‌ای استفاده می‌شود که برابر ۱ قرار گرفته‌است.

۲-۱-۵ مؤلفه‌ی یافتن نزدیک‌ترین همسایه

در مازول استخراج ویژگی به جهت ایجاد بردارهای ویژگی برای هر برنامه، از نرم‌افزار *apktool* به جهت انجام دیس‌اسمبل فایل‌های *apk* استفاده شده‌است. *apktool* ابزار قدرتمندی است که اجازه‌ی انجام عملیات مهندسی معکوس^{۱۰} و دستیابی به کد اصلی برنامه‌های اندرویدی را ممکن می‌سازد.

- پارامتر K : مقدار این پارامتر مشخص‌کننده‌ی تعداد همسایه‌های نزدیک به برنامه هدف می‌باشد که به جهت ارزیابی با توجه به مجموعه‌ی داده‌ی آزمون، مقدار برای ۲۵۰ برای آن در نظر گرفته شده‌است.

۳-۱-۵ مؤلفه‌ی مقایسه‌ی دودویی و تشخیص برنامه‌های بازبسته‌بندی شده

در این مولفه برای مقایسه‌ی امضای نهایی برنامه‌های اندرویدی با استفاده از چکیده‌سازی محلی، از سه کتابخانه‌ی پایتونی *Tlsh*، *Fuzzyhashlib*، *ssdeep* استفاده شده‌است.

- پارامتر Thr_c : این پارامتر حد بیشینه و یا کمینه را با توجه به روش مورد استفاده در قسمت چکیده‌سازی مشخص می‌کند. مقدار حد کمینه برای روش‌های چکیده‌سازی محلی *sdhash*، *ssdeep*، عدد ۷۰ و برای روش *tlsh* مقدار بیشینه‌ی ۶۰ در نظر گرفته شده‌است.

۲-۵ مجموعه داده‌ی آزمون

در قسمت تشخیص کدهای کتابخانه‌ای، با استفاده از جمع‌آوری کتابخانه‌های مشهور و استفاده از مخزن کتابخانه‌های پژوهش آقای وانگ [۵۳] (شامل ۴۵۱ کتابخانه) در نهایت مخزنی از ۸۷۷ کتابخانه‌ی اندرویدی ساخته شد. مخزن کتابخانه‌های مورد استفاده در پژوهش آقای ترکی [۵۲]، همان مخزن کتابخانه‌ای در پژوهش آقای وانگ می‌باشد. پس از بررسی و ارزیابی‌های اولیه‌ی استخراج کدهای کتابخانه‌ای در این پژوهش متوجه شدیم که بخشی از کتابخانه‌های اندرویدی شناسایی نمی‌شود. بنابراین با افزایش تعداد کتابخانه‌های مخزن، توانستیم مجموعه داده‌ی آزمون استفاده شده در این پژوهش را بهبود ببخشیم.

^{۱۰} Reverse Engineering

به جهت ارزیابی تشخیص برنامه‌های بازسته‌بندی شده، از داده‌ی آزمون جمع‌آوری شده در پژوهش اندروزو^{۱۱} [۹۷] استفاده شده است. این مجموعه داده‌ی آزمون شامل ۱۵۰۰۰ جفت بازسته‌بندی شده است که از مجموعه‌ی ۲۲ میلیون برنامه‌ی اندرویدی استخراج شده است. برای ساخت داده‌ی آزمون پژوهش، ۷۹۶ جفت بازسته‌بندی شده و ۴۰۰ جفت برنامه‌های غیر تقلبی از میان این مجموعه استفاده شده است. همچنین مجموعه‌ی داده‌ی آزمون پژوهش شامل ۱۱۸۱ برنامه‌ی اندرویدی است که جفت‌های تقلبی و غیر تقلبی را تشکیل می‌دهند.

۳-۵ ارزیابی و مقایسه

از آنجایی که هدف این پژوهش افزایش کارایی تشخیص برنامه‌های اندرویدی بازسته‌بندی شده است. بنابراین در این قسمت، ابتدا مقایسه‌ای از روش‌های چکیده‌سازی محلی از نظر زمان اجرا و دقت تشخیص، به صورت دودویی خواهیم داشت و دلایل بهبود این روش را نسبت به روش پیشین بررسی خواهیم کرد. در قسمت بعدی ایده‌ی اصلی پژوهش یعنی استفاده از طبقه‌بندی نزدیک‌ترین همسایه برای کاهش فضای مقایسه را مورد ارزیابی قرار می‌دهیم.

به جهت پیاده‌سازی الگوریتم‌های سه مولفه‌ی تشخیص کدهای کتابخانه‌ای، طبقه‌بندی نزدیک‌ترین همسایه و تشخیص برنامه‌های بازسته‌بندی از زبان پایتون استفاده شده است. همچنین به جهت مقایسه‌ی امضای کلاس‌ها و امضای برنامه‌های اندرویدی از سه کتابخانه‌ی *Tlsh*، *Fuzzyhashlib*، *ssdeep* در زبان پایتون استفاده شده است. ارزیابی‌های پژوهش در ماشینی با پردازنده‌ی *Corei5 - ۱۰۴۰۰* و *۳۲G* حافظه‌ی *Ram* اجرا شده است.

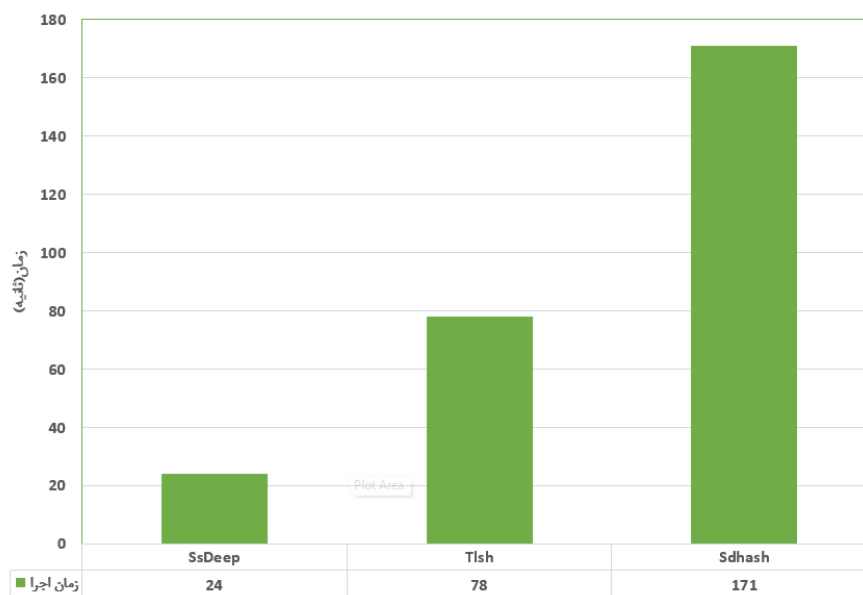
۱-۳-۵ مقایسه‌ی دودویی بدون طبقه‌بندی

با بررسی روش پیشین پیاده‌سازی شده در پژوهش آقای ترکی، دریافتیم که به صورت کلی این پژوهش را از دو جنبه‌ی متفاوت می‌توان بهبود بخشید. امضای متد در پژوهش آقای ترکی، حاوی مقادیر زیادی افزونگی ناشی از اضافه‌نمودن امضای کلاس در خروجی و ورودی متدها می‌باشد به همین علت طول امضا در برخی از برنامه‌های اندرویدی مخزن از نیم میلیون کاراکتر نیز فراتر رفته است. ایجاد افزونگی در امضای متد منجر به افزایش قابل توجه طول امضای کلاسی می‌شود.

به دلیل افزایش افزونگی در امضای کلاسی، تعداد کلاس‌های کاندید ناشی از اعمال الگوریتم‌های ساختاری و طولی افزایش یافته و موجب افزایش گره‌های گراف دوبخشی در حل مسأله‌ی تخصیص در ماژول نگاشت

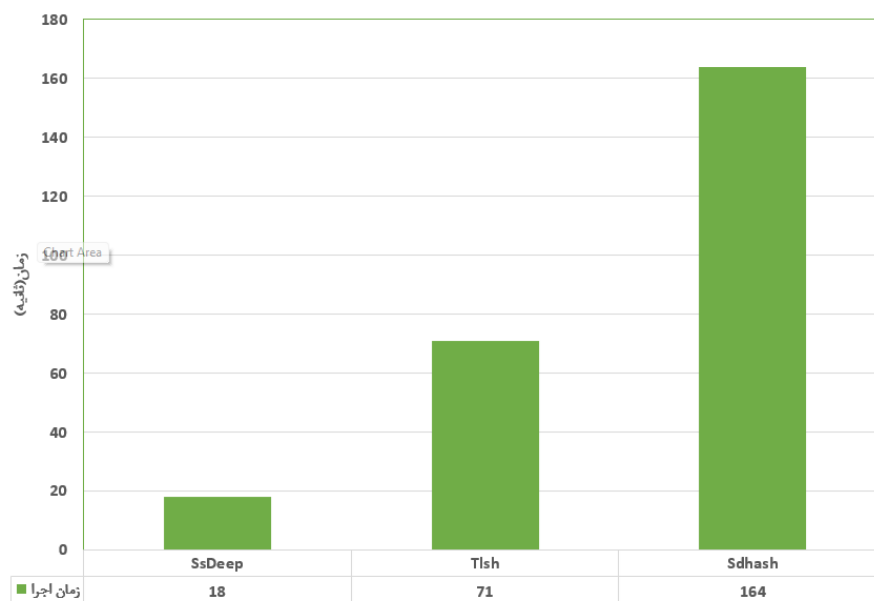
^{۱۱} Androzoo

کلاس‌های کتابخانه‌ای و برنامه‌ک می‌گردد. از آنجایی که حل مسئله‌ی تخصیص در گراف‌های دوبخشی از مرتبه‌ی زمانی n^3 است بنابراین افزایش تعداد گره‌های گراف دوبخشی حاصل منجر به کاهش سرعت قابل توجه در این پژوهش می‌شود. بنابراین در قسمت تشکیل امضا در این پژوهش، توانستیم با اضافه‌نمودن ویژگی‌های ثانویه و مورد نیاز، نظیر توابع فراخوانی‌شده در بدنه‌ی کلاس، تا حدودی افزونگی موجود در این پژوهش را مرتفع سازیم به طوری که ارزیابی حاصل از اعمال امضای جدید، دقت خوبی داشته‌باشد.



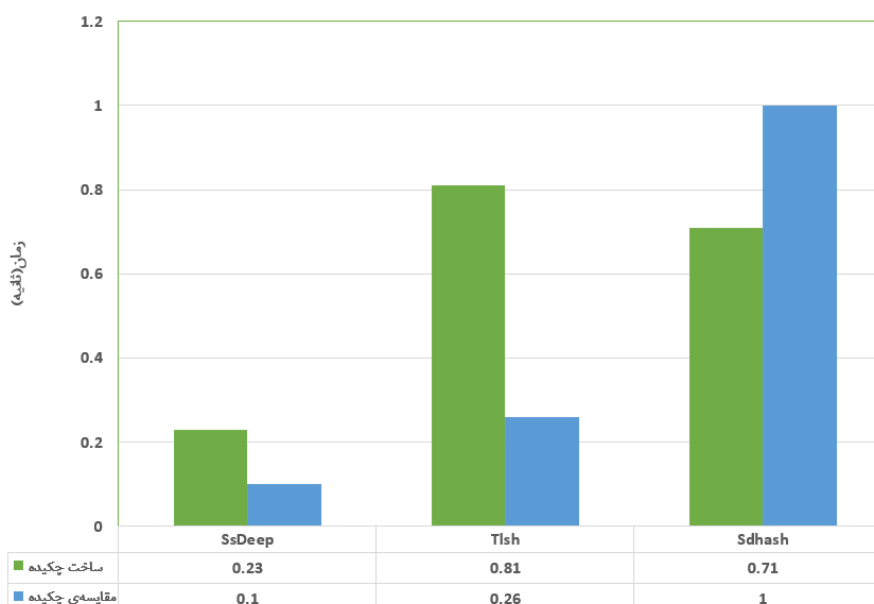
شکل ۵-۱: مقایسه‌ی میانگین زمان اجرای روال تشخیص برنامه‌های بازسته‌بندی شده

همانطور که در شکل ۵-۱ مشاهده می‌شود، با مقایسه‌ی میانگین زمان لازم برای اجرای تشخیص بازسته‌بندی از مرحله‌ی ابتدایی تشخیص کدهای کتابخانه‌ای تا انتها بر روی ۷۹۶ جفت بازسته‌بندی شده و ۴۰۰ جفت غیر تقلبی، به ازای ۳ روش‌های چکیده‌سازی محلی، می‌توان متوجه شد که روش *ssdeep* دقت بالاتری نسبت به باقی روش‌ها دارد چرا که میانگین زمان اجرای روال تشخیص بازسته‌بندی به ازای برنامه‌های داخل مخزن، از تمامی روش‌های دیگر بهتر است. از طرفی با مقایسه‌ی نمودارهای ۵-۲ و ۵-۴ مشخص می‌شود که عمده‌ی زمان اجرا مختص تشخیص کتابخانه‌های اندرویدی است چرا که در این مرحله از تشخیص، یک مسأله‌ی گرافی با مرتبه‌ی بالایی حل می‌شود. بنابراین می‌توان نتیجه‌گرفت که با بهبود امضای برنامه‌های اندرویدی و در نتیجه حذف افزونگی موجود در پژوهش ترکی، می‌توان سرعت تشخیص را به صورت قابل توجهی افزایش داد چرا که با حذف این افزونگی‌ها، گره‌های حاصل از گراف دوبخشی کاهش یافته و مسأله‌ی گرافی با سرعت بالاتری حل می‌شود.



شکل ۵-۲: مقایسه‌ی میانگین زمان اجرای مرحله‌ی تشخیص کدهای کتابخانه‌ای

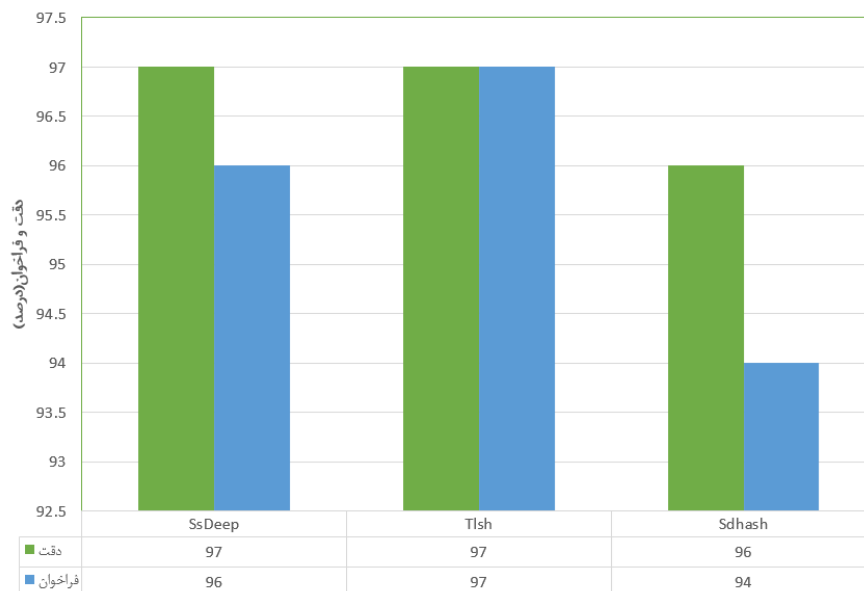
در نهایت با حذف افزونگی‌های امضا در پژوهش ترکی، توانستیم روش موجود را به ازای مقایسه‌ی دودویی برنامه‌های اندرویدی بهبود بخشیم. همانطور که در شکل ۵-۵ مشخص شده است، استفاده از امضای روش ترکی به دلیل افزونگی زیاد، در مرحله‌ی تشخیص کتابخانه‌های اندرویدی سرعت بسیار پایینی دارد.



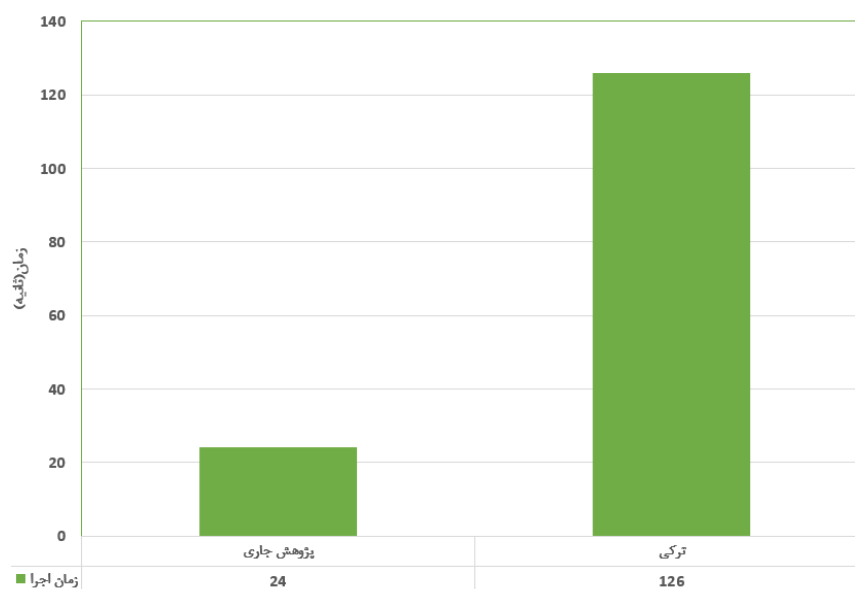
شکل ۵-۳: مقایسه‌ی میانگین زمان اجرای مراحل ساخت چکیده و مقایسه‌ی چکیده‌ها

مقایسه‌ی صورت‌گرفته میان روش این پژوهش و پژوهش ترکی با استفاده از تولید امضا و مقایسه‌ی

مبتنی بر چکیده‌سازی *Ssdeep* انجام شده‌است چرا که این روش، چه از نظر دقت مطابق شکل ۴-۳ جو عسکل: و چه سرعت نتیجه‌ی بهترین نسبت به باقی روش‌ها داشت. با مقایسه‌ی روش پیاده‌سازی شده و روش ترکی متوجه شدیم که در بخش تشخیص کتابخانه‌های اندرویدی، به طور میانگین ۳۱۵ گره کمتر از مرحله‌ی فیلتر کتابخانه‌ها عبور می‌کنند که این موجب می‌شود تعداد گره‌های گراف دوبخشی نهایی بسیار کمتر از روش آقای ترکی باشد.

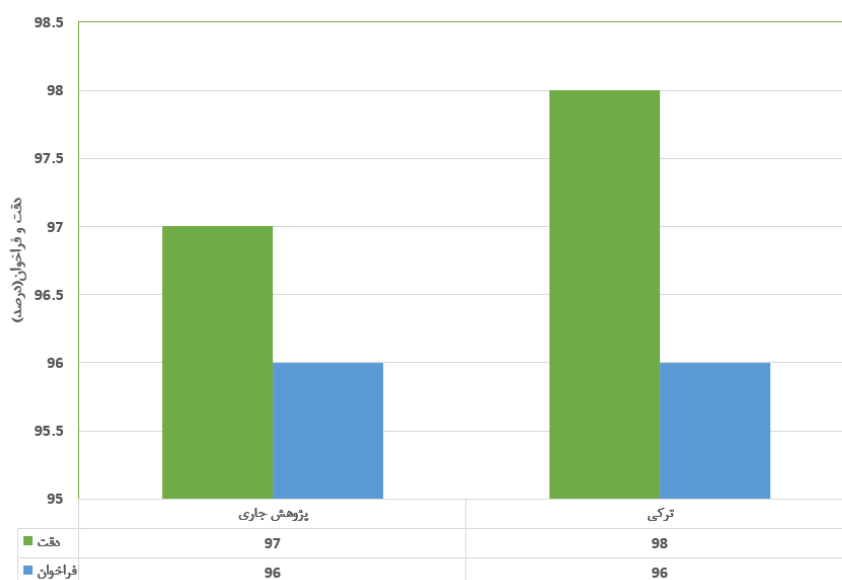


شکل ۴-۵: مقایسه‌ی دقت و فراخوان روش‌های چکیده‌سازی محلی در پژوهش جاری



شکل ۵-۵: مقایسه‌ی میانگین زمان اجرای مراحل تشخیص پژوهش جاری با پژوهش ترکی [۵۲]

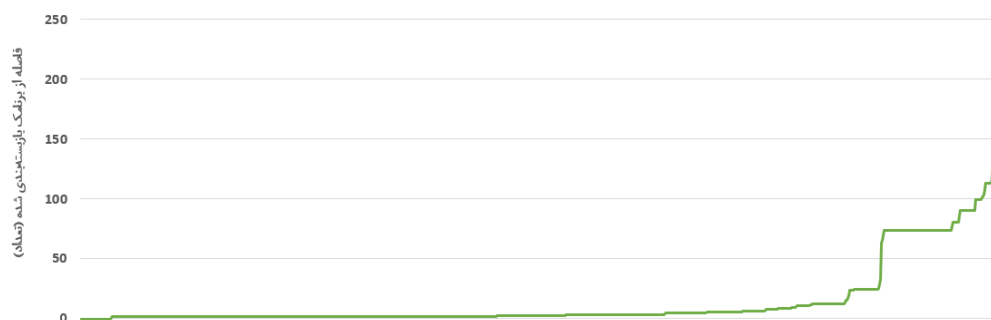
همچنین در مرحله‌ی ساخت امضای برنامه‌ک، به دلیل جست‌وجوی زیاد ناشی از افزونگی آن، تولید امضا در پژوهش جاری به صورت میانگین ۱۰ ثانیه به طور می‌انجامد. این در حالی است که با حذف افزونگی‌های امضا توانستیم زمان تشکیل امضای برنامه‌ک‌های اندرویدی را به ۴/۷ ثانیه کاهش دهیم. همانطور که مشاهده‌شد، با حذف افزونگی‌های امضای برنامه‌ک در پژوهش آقای ترکی توانستیم سرعت پژوهش اخیر را بهبود ببخشیم. به علاوه در ابتدای این فصل بیان کردیم که هدف از این پژوهش افزایش کارایی پژوهش‌های اخیر، با در نظر گرفتن معیارهای سرعت و دقت، بوده‌است. بنابراین قربانی نمودن دقت پژوهش در ازای دریافت سرعت بالا منطقی نمی‌باشد و ایجاد یک توازن میان این دو معیار، حیاتی است. بنابراین با حذف افزونگی‌های پژوهش ترکی، نیازمند اضافه نمودن ویژگی‌هایی بودیم که در کنار یکتا بودن در کلاس‌های برنامه‌ک، بتواند بخشی از کارکرد امضای کلاسی در مواقعی که متدهای کلاس دارای خروجی و ورودی‌های غیرجاوایی هستند را داشته‌باشد. با اضافه نمودن ویژگی‌های یکتا و تغییر امضای پژوهش ترکی، در کنار افزایش سرعت تشخیص برنامه‌ک‌های بازبسته‌بندی شده، خصوصا در بخش تشخیص کدهای کتابخانه‌ای، توانستیم دقت پژوهش جاری را به میزان خوبی بالا نگه‌داریم. با مقایسه‌ی برنامه‌ک‌های داخل مخزن داده‌ی آزمون، شامل ۱۱۸۱ جفت بازبسته‌بندی شده و غیر تقلبی، روش این پژوهش تنها در موارد مثبت غلط نسبت به پژوهش ترکی افت داشته‌است که موجب دقت پایین‌تری نسبت به آن شده‌است. این در حالی است که از نظر سرعت اجرا، بهبود ۶ برابری را در تشخیص مشاهده می‌کنیم.



شکل ۵-۶: مقایسه‌ی دقت و فراخوان پژوهش جاری و ترکی [۵۲]

۵-۳-۲ مقایسه‌ی دودویی همراه با طبقه‌بندی

در ابتدای بخش ۵-۳-۱ اشاره کردیم که پژوهش انجام‌شده توسط آقای ترکی را می‌توان از دوجنبه‌ی متفاوت بهبود بخشید. از منظر دوم، مقایسه‌ی تمامی برنامه‌های اندرویدی موجود در مخزن به ازای یک برنامه هدف ورودی، روشی زمان‌بر و بیهوده‌است چرا که بسیاری از برنامه‌های اندرویدی داخل مخزن هیچ شباهتی به برنامه ورودی ندارند. به همین جهت ما در این پژوهش از یک مرحله طبقه‌بندی به جهت کاهش فضای مقایسه‌ی دودویی پژوهش آقای ترکی در کنار امضای بهبود داده‌شده استفاده کرده‌ایم. پس از اعمال طبقه‌بند نزدیک‌ترین همسایه بر روی جفت اول و یا دوم از ۷۹۶ جفت برنامه بازبسته‌بندی شده، تنها ۱۹ جفت برنامه، خارج از بازه‌ی ۲۵° برنامه نزدیک به برنامه هدف قرار گرفته‌اند. به عبارت دیگر، پس از استخراج تمامی ۲۵° همسایه‌ی نزدیک به برنامه اندرویدی هدف، از میان جفت‌های بازبسته‌بندی شده، بیش از ۹۷٪ از آن‌ها، در بازه‌ی ۲۵° تایی از جفت تقلبی خود قرار گرفته‌اند. به علاوه، برای ۸۱٪ از جفت‌های بازبسته‌بندی شده، در هنگام اعمال طبقه‌بندی مبتنی بر نزدیک‌ترین همسایه، کمتر از ۲۰ برنامه با جفت تقلبی خود فاصله داشته‌اند. لازم به ذکر است، بازه‌ی ۲۰ تایی میان برنامه اصلی و برنامه تقلبی ممکن است شامل تعدادی از بسته‌های بازبسته‌بندی شده‌ی دیگری باشد که در لیست داده‌ی آزمون پژوهش که به صورت تصادفی انتخاب شده‌است نباشد. به علاوه، بررسی داده‌های اندروزی بر روی داده‌های ۲۲ میلیون برنامه اندرویدی، تمامی برنامه‌های بازبسته‌بندی شده را شامل نمی‌شود و تعدادی از برنامه‌ها در این لیست ۱۵۰۰۰ تایی آورده نشده‌اند. بنابراین، ارزیابی روش، به خصوص در مورد فاصله از جفت تقلبی، می‌تواند بسیار بهتر از ارزیابی فعلی باشد.



شکل ۵-۷: نمودار فاصله از جفت بازبسته‌بندی شده به ازای تمامی برنامه‌های بازبسته‌بندی شده

با استفاده از اجرای طبقه‌بندی بر روی برنامه‌های اندرویدی داخل مخزن، تعداد برنامه‌های مورد بررسی برای هر جفت از ۱۱۸۰ برنامه به ۲۵۰ برنامه نزدیک به خود کاهش پیدا کرد. ارزیابی روش ارائه‌شده مطابق نمودار ۵-۷ نشان می‌دهد که عملکرد طبقه‌بند نزدیک‌ترین همسایه، برای کاهش فضای مقایسه‌ی دودویی کاملاً موثر می‌باشد چرا که فضای مقایسه‌ی دودویی را ۸۰ درصد کاهش داده‌است.

علاوه بر این ۹۷٪ از جفت‌های بازبسته‌بندی شده در فاصله‌ی ۲۵۰ تا‌ی از برنامه‌ک هدف قرار گرفته‌اند که موجب می‌شود طبقه‌بندی و اجرای مقایسه‌ی دودویی روی همسایه‌ها نزدیک، با توجه به بهبود سرعت امضا، منجر به افزایش سرعت مقایسه‌ی دودویی در کنار حفظ دقت پژوهش‌های اخیر گردد. به‌جهت مقایسه‌ی زمانی پژوهش جاری و مقایسه‌ی دودویی معمول موجود در پژوهش آقای ترکی، فرض کنید که اجرا و مقایسه‌ی هر جفت برنامه‌ک مطابق با شکل ۵-۵ به ترتیب ۲۴ ثانیه و ۱۲۶ ثانیه زمان خواهد برد. در این صورت در مقایسه‌ی دودویی پژوهش صورت‌گرفته در روش آقای ترکی، برای یافتن جفت بازبسته‌بندی نیازمند صرف ۱۴۸۰۰۰ ثانیه زمان هستیم. این در حالی‌است که زمان میانگین مورد نیاز در پژوهش جاری، با توجه به میانگین فاصله‌ی برنامه‌ک از جفت بازبسته‌بندی شده که عدد ۱۶ می‌باشد، ۳۸۴ ثانیه‌است که اختلاف معناداری را به جهت مقایسه‌ی برنامه‌ک‌های اندرویدی ایجاد می‌کند.

۴-۵ تحلیل و جمع‌بندی عملکرد روش پیشنهادی

به طور کلی پژوهش جاری، پژوهش آقای وانگ [۵۳] و آقای ترکی [۵۲] را می‌توان در دسته‌ی روش‌های مبتنی بر تحلیل ایستا با استفاده از ویژگی‌های کدپایه به حساب آورد. هدف اصلی پژوهش وانگ، ارائه‌ی ابزاری به جهت تشخیص کدهای کتابخانه‌ای با استفاده از مقایسه‌ی دودویی ویژگی‌های کلاسی و یافتن کلاس‌های کتابخانه‌ای بوده‌است. استفاده از ایده‌ی ایجاد نگاشت میان کلاس‌های کتابخانه‌ای و کلاس‌های برنامه‌ک اولین بار در پژوهش آقای وانگ عنوان شده‌است و در ادامه توسط آقای ترکی با استفاده از فیلترهای طول امضا و ساختاری، بهبود پیدا کرده‌است. بنابراین پژوهش آقای وانگ را می‌توان سنگ‌بنای تشخیص کدهای کتابخانه‌ای در پژوهش ترکی و پژوهش جاری محسوب نمود. اما با توجه به تمرکز هر دو پژوهش، بهبود محسوسی در هیچ‌کدام از آن‌ها در قسمت تشخیص برنامه‌ک‌های بازبسته‌بندی شده مشاهده نشده‌است. هر دو پژوهش از همان ایده‌ی پرهزینه‌ی مطروحه در قسمت شباهت‌سنجی دودویی میان کلاس‌های برنامه‌ک و در نهایت شباهت‌سنجی کل برنامه‌ک‌های داخل مخزن با تمامی برنامه‌ک‌های دیگر استفاده کرده‌اند. علاوه بر این، از آن‌جایی که بسیاری از وابستگی‌های کلاسی نظیر وابستگی کلاس فرزند به کلاس پدر، کلاس‌های درونی و بیرونی و واسط‌های پیاده‌سازی شده، در پژوهش آقای ترکی به جهت افزایش دقت در تشخیص کدهای کتابخانه‌ای اضافه شده‌است بنابراین حجم امضای برنامه‌ک‌های اندرویدی در این پژوهش نسبت به پژوهش وانگ افزایش قابل توجهی داشته‌است زیرا امضای برنامه‌ک، جزئیات بیشتری از هر قسمت را با خود به همراه دارد. گرچه با استفاده از ایده‌ی فیلتر طول امضا در پژوهش آقای ترکی سعی شده تا قسمتی از این افزونگی داده حل شود اما همچنان مقایسه‌ی امضای کلاس‌ها در قسمت نگاشت میان کلاس‌های کتابخانه‌ای بسیار پرهزینه‌است. ایده‌ی دیگری که در پژوهش آقای ترکی استفاده شده‌است، تطابق امضای متدها در گراف فراخوانی میان متدها است که در پژوهش آقای وانگ به آن توجهی نشده‌است. استفاده از

گراف فراخوان متد، موجب می‌شود تا امضای متدهایی که به صورت ایستا مبهم شده‌اند و داخل گراف فراخوان نیستند ساخته نشود.

تمرکز این پژوهش بر روی بهبود کارایی در قسمت تشخیص برنامه‌های اندرویدی بازسته‌بندی شده بوده‌است. اگر چه بهبود امضای کلاس‌های برنامه و کاهش افزونگی‌های موجود در امضای کلاسی حاصل از پژوهش ترکی موجب شده‌است بخش تشخیص کدهای کتابخانه‌ای نیز بهبود پیدا کند اما حذف افزونگی‌های موجود در پژوهش ترکی با استفاده از جایگزینی ویژگی‌های طولانی با ویژگی‌های مختصر اما یکتا باعث شده‌است تا در قسمت مقایسه‌ی دودویی برنامه‌ها افزایش سرعت ۶ برابری داشته‌باشیم. علی‌رغم افزایش سرعت، همچنان ایده‌ی مقایسه‌ی دودویی یک برنامه با تمامی برنامه‌های داخل مخزن به جهت یافتن جفت تقلبی آن بیهوده و زمان‌بر به نظر می‌رسید. برای حل این مشکل، از یک ایده‌ی طبقه‌بندی استفاده شد. با استفاده از یک طبقه‌بند نزدیک‌ترین همسایه و ویژگی‌های مبتنی بر منابع برنامه‌های اندرویدی، مشکوک‌ترین برنامه‌ها را از داخل مخزن انتخاب و مقایسه‌ی آن‌ها را انجام می‌دهیم. طبقه‌بندی گرچه موجب کاهش یک درصدی دقت شده‌است، اما فضای مقایسه‌ی برنامه‌های اندرویدی را ۵ برابر کاهش داده که از نظر پژوهش مقرون به صرفه می‌باشد.

از نظر دسته‌بندی در روش‌های ایستا، پژوهش جاری را می‌توان ترکیبی از روش‌های مبتنی بر گراف و مبتنی بر منابع برنامه با استفاده از ویژگی‌های آن دانست. در این پژوهش برای تشکیل امضا از فراخوانی‌های موجود در گراف فراخوانی متد، واسط‌های برنامه‌نویسی و مجموعه‌ای از ویژگی‌های کدپایه استفاده شده‌است که تمامی آن‌ها در دسته‌ی تحلیل ایستا قرار می‌گیرند. از طرفی برای طبقه‌بندی از ویژگی‌های مبتنی بر منابع برنامه استفاده شده‌است چرا که منابع برنامه‌های اندرویدی تمایز درشت‌دانه میان برنامه‌ها برقرار می‌سازند.

فصل ۶

نتیجه‌گیری

به طور کلی روش‌های تشخیص برنامه‌های بازبسته‌بندی شده، برای تشخیص جفت بازبسته‌بندی شده باید برنامه‌های داخل مخزن را با برنامه هدف مقایسه کنند. استخراج ویژگی از برنامه‌های اندرویدی و مقایسه‌ی دوبه‌دوی آن‌ها با یکدیگر، ساختار اصلی اکثر پژوهش‌های موجود در این زمینه را تشکیل می‌دهد. تقریباً تمامی روش‌های مبتنی بر تحلیل ایستا برای مقایسه، سعی در استخراج مدلی از برنامه دارند که نشان‌دهنده‌ی مهم‌ترین ویژگی‌های ساختاری آن‌ها باشد. مقاومت روش ارائه‌شده در مقابل راهکارهای مبهم‌نگاری، به صورت مستقیم وابسته به مدلی است که روش پژوهش از آن برای نشان‌دادن برنامه استفاده می‌کند. بنابراین به هر میزان که ویژگی‌های منتخب یکتا باشند، مقاومت در مقابل مبهم‌نگاری و در نهایت دقت پژوهش نیز افزایش می‌یابد.

در کنار توجه به افزایش دقت در پژوهش‌های این حوزه، سرعت تشخیص نیز اهمیتی برابر دارد چرا که اگر ویژگی‌های منتخب و در نهایت مدل یکتای هر برنامه، ویژگی‌های متفاوتی را دربر داشته‌باشد و یا از روش‌های مبتنی بر گراف استفاده کند، آنگاه مقایسه‌ی مدل‌های موجود نیز سخت و زمان‌بر خواهد بود به طوری که در برخی از پژوهش‌های موجود در این حوزه، نظیر آنچه در پژوهش‌های مبتنی بر گراف دیده می‌شود، داده‌ی آزمون محدود به برنامه‌های کمی است که عملاً مقیاس‌پذیری ارزیابی ارائه‌شده را نقض می‌کند. از طرفی، پژوهش‌های موجود در زمینه‌ی تشخیص بازبسته‌بندی با استفاده از ویژگی‌های مبتنی بر منابع برنامه، گرچه سرعت بالایی را در تشخیص دارند اما از آنجایی که ایجاد مبهم‌نگاری در منابع، آسانتر از ویژگی‌های مبتنی بر کد برنامه است، روش‌های موجود در بهترین حالت می‌توانند راه‌حلی برای طبقه‌بندی درشت‌دانه را ارائه کنند و معمولاً در تشخیص جفت بازبسته‌بندی شده، دقت خوبی را ارائه نمی‌دهند.

در این پژوهش ابتدا با استفاده از مولفه‌ی تشخیص کدهای کتابخانه‌ای و به کمک ایده‌ی استفاده از فیلترهای

ساختاری و طولی، موجود در پژوهش ترکی [۵۲]، فضای مقایسه‌ای به جهت تشخیص کدهای کتابخانه‌ای را کاهش دادیم و به کمک ماژول نگاشت، پس از استفاده از توابع چکیده‌سازی محلی *sdhash*, *ssdeep*, *tlsh* و مقایسه‌ی کلاس‌های کتابخانه‌ای و کلاس‌های برنامه‌ی، لیستی از کلاس‌های کتابخانه‌ای کاندید را به ازای هر کلاس برنامه‌ی به دست آوردیم. در نهایت برای استخراج نگاشت میان کلاس‌های کتابخانه‌ای و کلاس‌های برنامه‌ی، مساله‌ی نگاشت میان کلاس‌های کتابخانه‌ای با استفاده از معکوس امتیاز تشابه را به مساله‌ی تخصیص در یک گراف دو بخشی تبدیل و در نهایت کلاس‌های کتابخانه‌ای برنامه‌ی را شناسایی می‌کنیم.

پس از استخراج کلاس‌های کتابخانه‌ای برنامه‌ی، حاصل از اعمال مولفه‌ی تشخیص کدهای کتابخانه‌ای، شروع به حذف آن‌ها کرده و کد اصلی برنامه‌ی را که توسط توسعه‌دهنده‌ی برنامه‌ی توسعه داده شده است را به دست می‌آوریم. در این قسمت ابتدا با استفاده از حذف افزونگی‌های پژوهش آقای ترکی و همچنین اضافه‌نمودن ویژگی‌های با طول کم‌تر، اندازه‌ی امضای نهایی برنامه‌ی هدف را به طور قابل توجهی کاهش دادیم. به طوری که در قسمت مقایسه‌ی دودویی برنامه‌ی، سرعت تشخیص پژوهش جاری در مقایسه با پژوهش آقای ترکی ۶ برابر و فراخوان پژوهش تنها ۱ درصد افت را تجربه کرده است.

از جهت دیگر، به جهت کاهش فضای مقایسه‌ی برنامه‌ی اندرویدی، ایده‌ی استفاده از یک طبقه‌بندی مبتنی بر نزدیک‌ترین همسایه با استفاده از ویژگی‌های مبتنی بر منابع مطرح شده است. استفاده از ویژگی‌های مبتنی بر منابع در کنار مقایسه‌ی دودویی با استفاده از ویژگی‌های مبتنی بر کد برنامه‌ی، منجر به مقاومت مناسب این روش در مقابل روش‌های مبهم‌نگاری شده است به طوری که از میان ۷۹۶ جفت بازبسته‌بندی شده تنها ۱۹ جفت در بازه‌ی ۲۵۰ تایی برنامه‌ی اندرویدی هدف قرار نگرفته‌اند که منجر به کاهش حدود ۵ برابری فضای مقایسه شده است.

به طور کلی، با مقایسه‌ی روش این پژوهش و پژوهش آقای ترکی، در قسمت مقایسه‌ی دودویی برنامه‌ی، با حفظ میزان دقت ۹۸٪ و کاهش ۱ درصدی فراخوان از ۹۸٪ به ۹۷٪ توانستیم سرعت روش این پژوهش را در حالت مقایسه‌ی دودویی ۶ برابر افزایش دهیم. علاوه بر این استفاده از طبقه‌بند مبتنی بر نزدیک‌ترین همسایه، دقتی معادل ۹۷٪ را در این پژوهش به همراه داشته است و در کنار آن، منجر به کاهش ۵ برابری فضای مقایسه‌ی برنامه‌ی اندرویدی شده است.

در انتها روش پیشنهادی در این پژوهش را می‌توان با استفاده از روش‌های متفاوت که در ادامه به آن‌ها اشاره شده است بهبود بخشید:

- یکی از ضعف‌های این پژوهش و اصولاً تمامی پژوهش‌های مبتنی بر تحلیل ایستا با استفاده از گراف جریان، اتکای ویژگی‌های آماری هر برنامه‌ی به گراف فراخوانی میان متدهای آن می‌باشد. پژوهش‌های موجود در مقابل اکثر روش‌های مبهم‌نگاری خصوصاً روش‌های مبهم‌نگاری که به صورت رایگان

در اختیار بدافزارنویسان هستند مقاوم هستند اما در صورتی که گراف فراخوانی برنامه دچار ابهام شود، تشخیص برنامه با استفاده از گراف فراخوانی باید با استفاده از روش‌های دیگر نظیر تحلیل پویای رفتار برنامه و بهبود سرعت آن، صورت بگیرد. از طرفی، تعدد برنامه‌های بازبسته‌بندی شده که به صورت رایگان مبهم شده‌اند نیز آنقدر زیاد است که مقابله‌ی با آن‌ها را حیاتی کرده‌است چرا که برنامه‌های تجاری خود به صورت ذاتی از روش‌های پیشگیری از بازبسته‌بندی استفاده می‌کنند.

- داده‌ی آزمون اکثر پژوهش‌های مطروحه در دسته‌ی تشخیص برنامه‌های بازبسته‌بندی شده عمومی نیستند و در بسیاری از آن‌ها حتی از ذکر جزئیات نحوه‌ی مبهم‌نگاری و یا فروض اولیه‌ی تعریف بازبسته‌بندی نیز خودداری شده‌است. به علاوه برنامه‌های تجاری نیز پیچیدگی‌های خاص خود را دارند که تشخیص بازبسته‌بندی در این دسته از برنامه‌ها را با مشکل مواجه می‌کند. ساخت یک داده‌ی آزمون مناسب شامل برنامه‌های تجاری و رایگان که بازبسته‌بندی شده‌اند، در کنار شفافیت در مورد چگونگی مبهم‌نگاری در آن‌ها می‌تواند به عنوان یک ایده‌ی پژوهشی مناسب در این زمینه مطرح شود.

- طبقه‌بندی در این پژوهش با اتکا به منابع ایستا در پوشه‌ی مربوط به واسط کاربری برنامه صورت می‌گیرد. تحلیل فراخوانی هر کدام از انواع منابع در کد برنامه‌های اندرویدی و در نهایت اجرای طبقه‌بندی روی ویژگی‌های مبتنی بر این دسته، می‌تواند به عنوان یک ایده برای افزایش دقت مرحله‌ی طبقه‌بندی استفاده شود.

مراجع

- [1] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC '14*, page 56–65, New York, NY, USA, 2014. Association for Computing Machinery.
- [2] M. S. Bhatt, H. Patel, and S. Kariya. A Survey Permission Based Mobile Malware Detection. *Int.J.Computer Technology and Applications*, 6(5):852–856, 2015.
- [3] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.
- [4] N. T. Cam, N. H. Khoa, T. T. An, N. P. Bach, and V.-H. Pham. Detect repackaged android applications by using representative graphs. In *2021 8th NAFOSTED Conference on Information and Computer Science (NICS)*, pages 102–106, 2021.
- [5] Global mobile OS market share 2022 | Statista — statista.com. <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/#:~:text=Android%20maintained%20its%20position%20as,the%20mobile%20operating%20system%20market>. [Accessed 02-Feb-2023].
- [6] Play Protect | Google Developers — developers.google.com. <https://developers.google.com/android/play-protect>. [Accessed 02-Feb-2023].
- [7] Decompile and modify an Android application | cylab.be — cylab.be. <https://cylab.be/blog/69/decompile-and-modify-an-android-application>. [Accessed 02-Feb-2023].

- [8] A. Dizdar. OWASP Mobile Top 10 Vulnerabilities and How to Prevent Them — brightsec.com. <https://brightsec.com/blog/owasp-mobile-top-10/>. [Accessed 02-Feb-2023].
- [9] D. J. Wu, C. H. Mao, T. E. Wei, H. M. Lee, and K. P. Wu. DroidMat: Android malware detection through manifest and API calls tracing. *Proceedings of the 2012 7th Asia Joint Conference on Information Security, AsiaJCIS 2012*, pages 62–69, 2012.
- [10] K. Khanmohammadi, N. Ebrahimi, A. Hamou-Lhadj, and R. Khoury. Empirical study of android repackaged applications. *Empirical Software Engineering*, 24(6):3587–3629, 2019.
- [11] T. Vidas and N. Christin. Sweetening android lemon markets: Measuring and combating malware in application marketplaces. *CODASPY 2013 - Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, 2011:197–207, 2013.
- [12] P. Maniriho, A. N. Mahmood, and M. J. M. Chowdhury. A study on malicious software behaviour analysis and detection techniques: Taxonomy, current trends and challenges. *Future Generation Computer Systems*, 130:1–18, 2022.
- [13] Z. Ma, H. Wang, Y. Guo, and X. Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 653–656, 2016.
- [14] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. F. Wang, and K. Zhang. *Understanding android obfuscation techniques: A large-scale investigation in the wild*, volume 254. Springer International Publishing, 2018.
- [15] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: Evaluating Android anti-malware against transformation attacks. *ASIA CCS 2013 - Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 329–334, 2013.
- [16] Trail: The reflection api the javax; tutorials — docs.oracle.com. <https://docs.oracle.com/javase/tutorial/reflect/index.html>. [Accessed 02-Feb-2023].
- [17] X. Zhang, F. Breitinger, E. Luechinger, and S. O’Shaughnessy. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Science International: Digital Investigation*, 39:301285, 2021.

- [18] ProGuard Manual: Home | Guardsquare — guardsquare.com. <https://www.guardsquare.com/manual/home>. [Accessed 02-Feb-2023].
- [19] Allatori Java Obfuscator — codedemons.net. <http://www.codedemons.net/allatori.html>. [Accessed 02-Feb-2023].
- [20] Y. Wang. Obfuscation-Resilient Code Detection Analyses for Android Apps. 2018.
- [21] L. Ardito, R. Coppola, S. Leonardi, M. Morisio, and U. Buy. Automated Test Selection for Android Apps Based on APK and Activity Classification. *IEEE Access*, 8:187648–187670, 2020.
- [22] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon. An investigation into the use of common libraries in android apps. 1:403–414, 2016.
- [23] N. Karankar, P. Shukla, and N. Agrawal. Comparative study of various machine learning classifiers on medical data. pages 267–271, 2017.
- [24] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. *ACM International Conference Proceeding Series*, 2014-Decem(December):56–65, 2014.
- [25] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. page 25–36, 2014.
- [26] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on Android markets. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7459 LNCS:37–54, 2012.
- [27] H. Gonzalez, N. Stakhanova, and A. A. Ghorbani. Droidkin: Lightweight detection of android apps similarity. *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, 152(January 2015):436–453, 2015.
- [28] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren. Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection. *IEEE Transactions on Information Forensics and Security*, 15(8):987–1001, 2020.
- [29] A. Salem. Stimulation and Detection of Android Repackaged Malware with Active Learning. 2015.

- [30] T. Nguyen, J. T. McDonald, W. B. Glisson, and T. R. Anel. Detecting repackaged android applications using perceptual hashing. *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2020-January:6641–6650, 2020.
- [31] F. Alswaina and K. Elleithy. Android malware family classification and analysis: Current status and future directions. *Electronics (Switzerland)*, 9(6):1–20, 2020.
- [32] F. Akbar, M. Hussain, R. Mumtaz, Q. Riaz, A. W. A. Wahab, and K.-H. Jung. Permissions-based detection of android malware using machine learning. *Symmetry*, 14(4), 2022.
- [33] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren. Android HIV: A study of repackaging malware for evading machine-learning detection. *IEEE Transactions on Information Forensics and Security*, 15:987–1001, 2020.
- [34] Q. Zhang, X. Zhang, Z. Yang, and Z. Qin. An efficient method of detecting repackaged android applications. pages 056 (4 .)–056 (4 .), 01 2014.
- [35] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, L. Xie, X. Sun, Y. Zhongyang, Z. Xin, B. Mao, L. Xie, D. Code, X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie. Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph To cite this version : HAL Id : hal-01370361 Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph. pages 0–14, 2016.
- [36] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. page 317–326, 2012.
- [37] A. Desnos. Android: Static analysis using similarity distance. pages 5394–5403, 2012.
- [38] B. B. Rad and M. Masrom. Metamorphic Virus Detection in Portable Executables Using Opcodes Statistical Feature. *International Journal on Advanced Science, Engineering and Information Technology*, 1(4):403, 2011.
- [39] B. B. Rad, M. Masrom, and S. Ibrahim. Opcodes histogram for classifying metamorphic portable executables malware. pages 209–213, 2012.
- [40] Q. Jerome, K. Allix, R. State, and T. Engel. Using opcode-sequences to detect malicious android applications. In *2014 IEEE International Conference on Communications (ICC)*, pages 914–919, 2014.

- [41] C. Liangboonprakong and O. Sornil. Classification of malware families based on n-grams sequential pattern features. pages 777–782, 2013.
- [42] Y. D. Lin, Y. C. Lai, C. H. Chen, and H. C. Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *Computers and Security*, 39(PART B):340–350, 2013.
- [43] P. Faruki, V. Laxmi, V. Ganmoor, M. Gaur, and A. Bharmal. Droidolytics: Robust feature signature for repackaged android apps on official and third party android markets. In *2013 2nd International Conference on Advanced Computing, Networking and Security*, pages 247–252, 2013.
- [44] J. Ko, H. Shim, D. Kim, Y.-S. Jeong, S.-j. Cho, M. Park, S. Han, and S. B. Kim. Measuring similarity of android applications via reversing and k-gram birthmarking. page 336–341, 2013.
- [45] S. Kishore, R. Kumar, and S. Rajan. *Towards Accuracy in Similarity Analysis of Android Applications*, volume 11281 LNCS. Springer International Publishing, 2018.
- [46] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang. Plagiarizing smartphone applications: Attack strategies and defense techniques. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7159 LNCS:106–120, 2012.
- [47] J. Crussell, C. Gibler, and H. Chen. Scalable Semantics-Based Detection of Similar Android Applications. *Esorics*, pages 182–199, 2013.
- [48] W. Hu, J. Tao, X. Ma, W. Zhou, S. Zhao, and T. Han. Migdroid: Detecting app-repackaging android malware via method invocation graph. In *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*, pages 1–7, 2014.
- [49] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of “piggybacked” mobile applications. *CODASPY 2013 - Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, pages 185–195, 2013.
- [50] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. *Proceedings - International Conference on Software Engineering*, (1):175–186, 2014.
- [51] J. Zheng, K. Gong, S. Wang, Y. Wang, and M. Lei. Repackaged apps detection based on similarity evaluation. pages 1–5, 2016.

- [52] M. Torki. Detecting repackaged android applications. 2018.
- [53] Y. Wang. Obfuscation-resilient code detection analyses for android apps,” phd thesis, the ohio state university. 2018.
- [54] X. Wu, D. Zhang, X. Su, and W. Li. Detect repackaged android application based on http traffic similarity. *Sec. and Commun. Netw.*, 8(13):2257–2266, sep 2015.
- [55] M. Alshehri. APP-NTS: a network traffic similarity-based framework for repacked Android apps detection. *Journal of Ambient Intelligence and Humanized Computing*, 13(3):1537–1546, 2022.
- [56] G. He, B. Xu, L. Zhang, and H. Zhu. On-device detection of repackaged android malware via traffic clustering. *Security and Communication Networks*, 2020:8630748, May 2020.
- [57] A. Rodriguez and A. Laio. Clustering by fast search and find of density peaks. *Science*, 344(6191):1492–1496, 2014.
- [58] J. Malik and R. Kaushal. Credroid: Android malware detection by network traffic analysis. In *Proceedings of the 1st ACM Workshop on Privacy-Aware Mobile Computing*, PAMCO ’16, page 28–36, New York, NY, USA, 2016. Association for Computing Machinery.
- [59] D. Iland and A. Pucher. Detecting android malware on network level. 2011.
- [60] S. Kandukuru and R. M. Sharma. Android malicious application detection using permission vector and network traffic analysis. pages 1126–1132, 2017.
- [61] M. Lin, D. Zhang, X. Su, and T. Yu. Effective and scalable repackaged application detection based on user interface. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6, 2017.
- [62] S. Yue, Q. Sun, J. Ma, X. Tao, C. Xu, and J. Lu. Regiondroid: A tool for detecting android application repackaging based on runtime ui region features. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 323–333, 2018.
- [63] Y. Hu, G. Xu, B. Zhang, K. Lai, G. Xu, and M. Zhang. Robust app clone detection based on similarity of ui structure. *IEEE Access*, 8:77142–77155, 2020.
- [64] S. Li. Juxtapp and dstruct: Detection of similarity among android applications. 2012.

- [65] J. Guo, D. Liu, R. Zhao, and Z. Li. Wltdroid: Repackaging detection approach for android applications. In G. Wang, X. Lin, J. Hendler, W. Song, Z. Xu, and G. Liu, editors, *Web Information Systems and Applications*, pages 579–591, Cham, 2020. Springer International Publishing.
- [66] F. Lyu, Y. Lin, J. Yang, and J. Zhou. Suidroid: An efficient hardening-resilient approach to android app clone detection. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 511–518, 2016.
- [67] X. Liu, Z. Yu, Z. Song, L. Chen, and Z. Qin. Mdsdroid: A multi-level detection system for android repackaged applications. In *2021 IEEE 6th International Conference on Signal and Image Processing (ICSIP)*, pages 1128–1133, 2021.
- [68] S. Yue, W. Feng, J. Ma, Y. Jiang, X. Tao, C. Xu, and J. Lu. Repdroid: An automated tool for android application repackaging detection. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 132–142, 2017.
- [69] J. Ma, Q.-W. Sun, C. Xu, and X.-P. Tao. Griddroid—an effective and efficient approach for android repackaging detection based on runtime graphical user interface. *Journal of Computer Science and Technology*, 37(1):147–181, Feb 2022.
- [70] V. Costamagna, C. Zheng, and H. Huang. Identifying and evading android sandbox through usage-profile based fingerprints. page 17–23, 2018.
- [71] Y. Zhang, K. Huang, Y. Liu, K. Chen, L. Huang, and Y. Lian. Timing-based clone detection on android markets. 12 2015.
- [72] T. Nguyen, J. T. McDonald, W. B. Glisson, and T. R. Aniel. Detecting repackaged android applications using perceptual hashing. 2020.
- [73] C. Soh, H. B. Kuan Tan, Y. L. Arnatovich, and L. Wang. Detecting clones in android applications through analyzing user interfaces. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 163–173, 2015.
- [74] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *2010 5th International Conference on Malicious and Unwanted Software*, pages 55–62, 2010.
- [75] D. Kim, A. Gokhale, V. Ganapathy, and A. Srivastava. Detecting plagiarized mobile apps using API birthmarks. *Automated Software Engineering*, 23(4):591–618, 2016.

- [76] Q. Guan, H. Huang, W. Luo, and S. Zhu. Semantics-based repackaging detection for mobile apps. In J. Caballero, E. Bodden, and E. Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 89–105, Cham, 2016. Springer International Publishing.
- [77] H. Wang, Y. Guo, Z. Ma, and X. Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 71–82, New York, NY, USA, 2015. Association for Computing Machinery.
- [78] O. S. J. Nisha and S. M. S. Bhanu. Detection of repackaged android applications based on apps permissions. In *2018 4th International Conference on Recent Advances in Information Technology (RAIT)*, pages 1–8, 2018.
- [79] W. Zhou, X. Zhang, and X. Jiang. Appink: Watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS ’13, page 1–12, New York, NY, USA, 2013. Association for Computing Machinery.
- [80] Y. Zhang and K. Chen. Appmark: A picture-based watermark for android apps. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 58–67, 2014.
- [81] C. Ren, K. Chen, and P. Liu. Droidmarking: Resilient software watermarking for impeding android application repackaging. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE ’14, page 635–646, New York, NY, USA, 2014. Association for Computing Machinery.
- [82] X. Sun, J. Han, H. Dai, and Q. Li. An active android application repacking detection approach. In *2018 10th International Conference on Communication Software and Networks (ICCSN)*, pages 493–496, 2018.
- [83] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu. Repackage-proofing android apps. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 550–561, 2016.
- [84] Q. Zeng, L. Luo, Z. Qian, X. Du, and Z. Li. Resilient decentralized android application repackaging detection using logic bombs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 50–61, New York, NY, USA, 2018. Association for Computing Machinery.

- [85] S. Tanner, I. Vogels, and R. Wattenhofer. Protecting android apps from repackaging using native code. In A. Benzekri, M. Barbeau, G. Gong, R. Laborde, and J. Garcia-Alfaro, editors, *Foundations and Practice of Security*, pages 189–204, Cham, 2020. Springer International Publishing.
- [86] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl. A large scale investigation of obfuscation use in google play. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC '18*, page 222–235, New York, NY, USA, 2018. Association for Computing Machinery.
- [87] Virus Bulletin :: Obfuscation in Android malware, and how to fight back — virusbulletin.com. <https://www.virusbulletin.com/virusbulletin/2014/07/obfuscation-android-malware-and-how-fight-back>. [Accessed 12-Feb-2023].
- [88] Y. Wang and A. Rountev. Who changed you? obfuscator identification for android. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 154–164, 2017.
- [89] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi. Droidnative: Automating and optimizing detection of android native code malware variants. *Computers and Security*, 65:230–246, 2017.
- [90] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, and F. Mercaldo. Detection of obfuscation techniques in android applications. In *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018*, New York, NY, USA, 2018. Association for Computing Machinery.
- [91] M. D. Preda and F. Maggi. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques*, 13:209–232, 2017.
- [92] Z. Li, J. Sun, Q. Yan, W. Srisa-an, and Y. Tsutano. Obfusifier: Obfuscation-resistant android malware detection system. In S. Chen, K.-K. R. Choo, X. Fu, W. Lou, and A. Mohaisen, editors, *Security and Privacy in Communication Networks*, pages 214–234, Cham, 2019. Springer International Publishing.
- [93] L. Zhang, H. Meng, and V. L. L. Thing. Progressive control flow obfuscation for android applications. In *TENCON 2018 - 2018 IEEE Region 10 Conference*, pages 1075–1079, 2018.
- [94] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC*

- Symposium on Information, Computer and Communications Security, ASIA CCS '13*, page 329–334, New York, NY, USA, 2013. Association for Computing Machinery.
- [95] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.
 - [96] W. Hou, D. Li, C. Xu, H. Zhang, and T. Li. An advanced k nearest neighbor classification algorithm based on kd-tree. In *2018 IEEE International Conference of Safety Produce Informatization (IICSPI)*, pages 902–905, 2018.
 - [97] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16*, page 468–471, New York, NY, USA, 2016. Association for Computing Machinery.

واژه‌نامه

الف

experimental	تجربی	heuristic	ابتنکاری
density	تراکم	high dimensions	ابعاد بالا
approximation	تقریب	bias	اریب
partition	تقسیم‌بندی	threshold	آستانه
mesh	توری	pigeonhole principle	اصل لانه‌ی کبوتری
distributed	توزیع‌شده	NP-Hard	ان‌پی-سخت
		transition	انتقال

ج

separable	جدایپذیر
black box	جعبه سیاه
data stream	جویبار داده

ح

extreme	حدی
greedy	حریصانه

خ

cluster	خوشه
linear	خطی

د

data	داده
data mining	داده‌کاوی

ب

online	برخط
linear programming	برنامه‌ریزی خطی
optimum	بهینه
maximum	بیشینه

پ

outlier	پرت
query	پرسمان
cover	پوشش
complexity	پیچیدگی

ت

	داده‌ی پرت outlier data
	دو برابر سازی doubling
	دودویی binary
ق	
deterministic قطعی	
ک	ر
efficient کارا	vertex رأس
candidate کاندیدا	formal رسمی
minimum کمینه	
م	ز
set مجموعه	sublinear زیرخطی
coreset مجموعه هسته	س
planar مسطح	amortized سرشکن
parallelization موازی سازی	hierarchichal سلسه مراتبی
buffer میان گیر	
ن	ش
inversion نابه جایی	pseudocode شبه کد
invariant ناورد	object شیء
center point نقطه‌ی مرکزی	
half space نیم فضا	ص
ه	غ
price of anarchy (POA) هزینه‌ی آشوب	dominate غلبه
ی	ف
edge یال	distance فاصله
	space فضا

پیوست آ

مطالب تکمیلی

Abstract

We present a standard template for typesetting theses in Persian. The template is based on the `XYLATEX Persian` package for the `LATEX` typesetting system. This write-up shows a sample usage of this template.

Keywords: Thesis, Typesetting, Template, `XYLATEX Persian`



Sharif University of Technology
Department of Computer Engineering

M.Sc. Thesis

Performance Improvement of Android Repackaged Applications

By:

Mojtaba Moazen

Supervisor:

Dr. Amini

february 2023